A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins

Christopher B. Walton Alfred G. Dale Roy M. Jenevein
Department of Computer Sciences
University of Texas at Austin

Abstract

Recent work on parallel joins and data skew has concentrated on algorithm design without considering the causes and characteristics of data skew itself. Existing analytic models of skew do not contain enough information to fully describe data skew in parallel implementations. Because the assumptions made about the nature of skew vary between authors, it is almost impossible to make valid comparisons of parallel algorithms. In this paper, a taxonomy of skew effects is developed, and a new performance model is introduced. The model is used to compare the performance of two parallel join algorithms.

1 Introduction

As databases expand and hardware becomes less expensive, research interest in parallel architectures and algorithms for relational processing has increased. Most research has focused on shared-nothing architectures[Stonebraker 86]. These systems consist of computational *nodes* with CPU, memory and secondary storage; nodes communicate exclusively by message passing. Because there is no single resource that must be shared by all nodes, it is possible to construct very large systems. Notable projects include GAMMA [DeWitt 88], Bubba[Copeland 88], Teradata[Teradata 83], and several varieties of hypercube[Frieder 90].

There has been considerable interest in parallel joins. Specific algorithms have been proposed in [Kitsuregawa 83, Gerber 86, Baru 87], while more general analyses and comparisons are presented in [Richardson 87, Boral 88, Lakshimi 88, Hu 89, Schneider 89,90]. Nearly all of this work makes *uniformity assumptions*: tuples are uniformly distributed to nodes at every stage of the join, and all join key values occur with equal frequency. Under these ideal conditions, parallel join algorithms are quite scalable [DeWitt 86, Gerber 87]. That is, system performance is roughly proportional to the number of nodes.

However, there is considerable evidence that data skew – the non-uniform distribution of tuples and key values – exists[Christodoulakis 83, Montgomery 83, Lynch 88]. The few published analyses of of joins in the presence of data skew, such as [Lakshimi 88, 89] indicate that data skew can curtail scalability. Recently, several parallel joins algorithms that are designed to minimize data skew have been proposed[Omiecinski 89, Wolf 89,90].

However, because existing models of data skew are inadequate, it is very difficult to make meaningful comparisons or evaluations of of these new algorithms. Present models are deficient in two ways: they do not recognize that data skew is a heterogeneous phenomenon, and they do not incorporate enough information to fully describe skewed data distributions in parallel systems.

This paper addresses both of these shortcomings, and demonstrates how an improved model of data skew can be used to evaluate algorithm performance. It is organized as follows: Section 2 introduces a taxonomy of data skew effects, and uses it to classify previous work on data skew. Section 3 examines shortcomings in previous models of data skew. A new model, the vector relative partition model, is introduced. Next, Section 4 explains how to use the vector relative partition model to analyze algorithm performance. In section 5, as an illustration of the model and the method, the GAMMA Hybrid Hash[DeWitt 90], and the Scheduling Hash [Wolf 90] parallel join algorithms are examined. Section 6 discusses the results of this analysis. Finally, Section 7 offers some conclusions.

2 Classifying Data Skew Effects

The term "data skew" encompasses several related but distinct effects. In order to accurately evaluate the performance of parallel join algorithms in the presence of skew, one must take these differences into account. The most fundamental distinction is that between intrinsic skew and partition skew.

Intrinsic skew occurs when attribute values are not distributed uniformly. Hence, it can also be called *attribute value skew* (AVS). It is a property of the data and does not vary between join algorithm implementations. AVS may occur on both single and multiple node systems. A join of two relations with AVS may have a greater join selectivity and therefore a larger join product, compared to a join of two uniformly distributed relations with the same cardinality. There is no way a parallel implementation can avoid this extra workload. Rather, the challenge in algorithm design is to balance the workload between nodes when AVS occurs.

Partition skew occurs on parallel implementations when the workload is not balanced between nodes. Partition skew can occur only on multiple node systems, and its effects vary between implementations. In particular, some types of partition skew can occur even when the input data are uniformly distributed. Partition skew is described in more detail in section 2.2

2.1 Relationship between Intrinsic and Partition Skew

When a parallel join algorithm is applied to relations with intrinsic skew, the distribution of tuples between nodes is not a simple partition of the intrinsic distribution. The actual distribution of tuples is determined by the interaction of several factors:

- Characteristics of the parallel join implementation, including the hash function(s) or other mechanism used to partition input relations.
- The join query, especially selection predicates.
- Intrinsic skew (if any) in the input relations.

In other words, even if there is a simple function that characterizes AVS in the relation as a whole, this function is of little use in predicting partition skew and algorithm performance. Because of the fine (tuple-level) granularity at which both data and implementation must be modeled, predicting partition skew as a function of intrinsic skew is very difficult

A complete representation of AVS requires enumerating all pairs (v, n) where v is the join key value and n is the number of tuples with that key value. Since this is too much information to deal with analytically, one must approximate AVS by a function $f(v)$ such that $n \approx f(v)$.

Finding the number of tuples requires a summation of $f(v)$. In the single node case, this is a straightforward summation over all v. But the multiple node case requires summing $f(v)$ over some arbitrary subset of v, determined by the hash functions and other components of the algorithm. Since hash functions tend to "scatter" tuples with similar key values over many hash buckets, this summation is much more difficult to evaluate.

2.2 Types of Partition Skew

There are several types of partition skew, depending on where in the join algorithm the load imbalance occurs. Parallel join algorithms may be divided into four stages:

1. Tuples are retrieved from disk.
2. selection and projection predicates are applied.
3. tuples are partitioned and redistributed.
4. partitions are joined on each node.

Partition skew may occur at each stage. Thus, four types of skew may be observed:

Tuple Placement Skew (TPS) The initial distribution of tuples varies between partitions. For example, tuples may be partitioned by clustering attribute, in user specified ranges.

Selectivity Skew (SS) This occurs when the selectivity of selection predicates varies between nodes. A selection predicate that includes a range selection on the partitioning attribute is an obvious example.

Redistribution Skew (RS) Redistribution skew occurs when there is a mismatch between the distribution of join key values in a relation and the distribution expected by the redistribution mechanism (typically a hash function).

Join Product Skew (JPS) Join Product Skew occurs when the join selectivity at each node differs. It is a property of a *pair* of relations. As such, it is not manifested until the relations are joined.

At this point, it is instructive to use this taxonomy to review some of the previous work on data skew in parallel joins. It can be seen that various authors make a variety of assumptions about the nature of data skew:

- limited experiments on the effects of AVS have been performed on the GAMMA database machines[Schneider 89,90]. The input relations consisted of synthetic data with a normal distribution of join key values. These tests explicitly assume that TPS and SS are absent. From the published descriptions, it is impossible to evaluate RS and JPS.

- Wolf, Dias, and Yu propose and evaluate skew resistant sort-merge join algorithms [Wolf 89] and hash-based join algorithms [Wolf 90]. In both of these cases, AVS is modeled by a Zipf distribution. As with the GAMMA tests, it is assumed that neither TPS nor SS are present. However, in [Wolf 90], RS and JPS are (implicitly) considered.
- Analytic studies by Lakshimi and Yu [Lakshimi 88,89] consider the effects of TPS.
- Algorithms developed by Baru and Frieder [Baru 87,89, Frieder 90], exploit the hypercube communications architecture to perform dynamic data redistribution before joins, preventing RS. The design of the algorithm does not address SS or JPS. An assumption that join inputs are fully memory resident implies that TPS is absent.

3 Modeling Data Skew in Parallel Joins

To date, analytic work on data skew has used what might be termed *scalar* models. The multicomputer system on which the join is performed and the input relations are described by scalar quantities, such are cardinality and memory size. These scalar parameters are adequate to model a single node system or a parallel system with no data skew.

To model partition skew, such models are supplemented with additional scalar quantities that describe some type of data skew; however this approach requires making restrictive assumption about the way tuples are distributed between nodes.

While scalar models can demonstrate the adverse affects of data skew on performance and scalability, they cannot describe all possible ways that data could be partitioned between nodes. A completely general description requires one variable for each node. In other words, skew must be expressed as a *vector* quantity. Such a model is presented in the next section.

3.1 The Vector Relative Partition Model

The vector relative partition model retains the scalar quantities that describe the system and the data as whole, such as cardinality, and augments them with a *skew vector* of node-specific coefficients. For example, the number of tuples at each node is specified in terms of the average number of tuples per node, and a vector element that specifies the per node cardinality relative to this average. This representation separates the size of a relation from its distribution. Also, there is a single normalization constraint for each skew vector.

3.2 Operational Definition of Skew Vectors

The parameters of the vector relative partition model are defined in terms of tuple counts at various points of a join. Let N be the number of nodes. The following counts are required:

M1(i,R) the number of tuples in relation R initially stored on node i.

M2(i,R) the number of tuples in relation R remaining on node i after local selection.

M3(i,R) the number of tuples in relation R on node i after redistribution.

M4(i,R,S) the size of the local join product on i between relations R and S.

Given these counts, skew vectors may defined as in the following sections:

3.2.1 Tuple Placement skew

The mean partition size K^R is the average number of tuples initially stored at each node.

$$K^R \equiv \frac{1}{N}\sum_i M1(i,R) \tag{1}$$

The TPS skew vector $\mathbf{T}_i^R$ expresses the number of tuples at each node as a multiple of K^R. That is:

$$\mathbf{T}_i^R \equiv \frac{M1(i,R)}{K^R} \tag{2}$$

3.2.2 Selectivity Skew

The relation selectivity $\hat{\alpha}^R$ is the fraction of tuples in relation R that remain after local selection:

$$\hat{\alpha}^R = \frac{1}{NK^R}\sum_i M2(i,R) \tag{3}$$

Recall that NK^R is the initial cardinality, and $\sum_i M2(i,R)$ is the cardinality after selection. The *local selectivity* is the selectivity observed at a specific node. The selectivity skew vector expresses local selectivity as a multiple of the relation selectivity $\hat{\alpha}^R$:

$$\mathbf{S}_i^R \equiv \frac{1}{\hat{\alpha}^R}\frac{M2(i,R)}{M1(i,R)} \tag{4}$$

3.2.3 Redistribution Skew

The redistribution skew vector $\mathbf{R}_i^R$ indicates what fraction of the the relation R is placed on each node after the redistribution phase:

$$\mathbf{R}_i^R \equiv \frac{M3(i,R)}{\sum_i M3(i,R)} = \frac{M3(i,R)}{\hat{\alpha}^R N K^R} \qquad (5)$$

3.2.4 Join Product Skew

The join selectivity is the ratio of the join output cardinality to the product of the cardinalities of the input relations.

$$\hat{\rho}^{RS} \equiv \frac{\sum_i M4(i,R,S)}{\sum_i M2(i,R) \sum_i M2(i,S)} \qquad (6)$$

The JPS skew vector describes the ratio of output cardinality and the product of the cardinalities of the input partitions in terms of the join selectivity:

$$\mathbf{J}_i^{RS} \equiv \frac{M4(i,R,S)}{N\hat{\rho}^{RS} M3(i,R) M3(i,S)} \qquad (7)$$

The factor of N in the denominator is required for normalization. While there are N join partitions, the product of the whole relations is N^2 times larger than the product of the (join input) partitions.

4 Using the Model

The vector relative partition model can be used in two ways. If tuple counts from actual or simulated joins are known, they can be used to quantify partition data skew. A second approach is to use measured or estimated values of the skew vectors to estimate algorithm response time. Such estimates could be used in query optimization or to evaluate design alternatives for new algorithms. Response time calculation are explained in the next section.

4.1 Calculating Response Time

The following procedure is used to calculate the response time for a single join.

1. Decompose each algorithm into phases. All nodes must finish a phase before any node can start the next phase. That is, synchronization barriers between phases are identified.

2. Within each phase, identify the processing steps, and determine the resource (CPU, disk or communications) used.

3. Calculate the number of tuples involved in each step, based on data characteristics and the skew vectors.

4. For each step, calculate the processing time for that step from the number of tuples. Add the processing time to the total processing time for the resource used at that step.

5. When all steps have been examined, determine the most heavily used resource (bottleneck) for each node. The processing time for that resource determines the response time for that node. The response time for the phase is the response time of the slowest node.

6. Response time for the algorithm is the sum of the response times for each phase.

In step 4 of the above procedure, the number of disk and communications I/O operations must be calculated from the number of tuples. It is assumed that tuples are not split across messages or disk tracks. For notational convenience, we define the function Θ as the number of I/O operations required to process n tuples of length L with a buffer size of b:

$$\Theta(n,b,L) \equiv \left\lceil \frac{n}{\lfloor b/L \rfloor} \right\rceil \qquad (8)$$

4.2 System and Data Characteristics

Before proceeding to a detailed analysis of the two algorithms, we describe the system and data characteristics assumed in this study. This description will also serve to introduce our notation. Most parameter values are based on the *JoinABprime* test case of the Wisconsin benchmark[Dewitt 90]. They are summarized in Table 1:

	parameter	value
$\lVert S \rVert$	inner cardinality	10^6
$\lVert R \rVert$	outer cardinality	10^7
L^S	S-tuple length	208
L^R	R-Tuple length	208
$\hat{\alpha}$	selectivity	1.0
$\hat{\rho}^{RS}$	join selectivity	10^{-7}
M	memory capacity	8 Mb
N	number of nodes	32
D	disk track	8 Kb
m	message	8 Kb
U	mem. utilization	0.9

Table 1: system and data characteristics

The processing times for various operations are listed in Table 2

t_{disk}	read/write a disk page	20 msec
t_{hash}	hash tuple	3 μsec
t_{send}	send message	5 msec
t_{probe}	probe hash table	6 μsec
t_{recv}	receive message	5 msec
t_{join}	join output tuple	40 μsec
t_{filter}	selection	15 μsec
$t_{schedule}$	scheduling step	2 msec

Table 2: operation times

5 Comparing Parallel Join Algorithms

This section presents detailed analyses of the GAMMA Hybrid Hash [DeWitt 90, Schneider 89] algorithm and the IBM Scheduling Hash algorithm[Wolf 90]. The analysis of each algorithm phase is presented in a tabular format. For example, Table 3 describes the first phase of the Hybrid Hash algorithm. The first column describes the processing step. The second column specifies the number of tuples processed; this may be less than the number of tuples stored on the node. If a given skew type affects that step, then the corresponding skew vector appears in the second column. The last two columns give the resource (CPU, disk, communications) used, and the processing time for the step.

5.1 Analysis of the GAMMA Hybrid Hash

The Hybrid Hash algorithm has three phases. They are:

1. Partition the smaller (S) relation into B buckets. Concurrently build a hash table with tuples in the first bucket S_0.

2. Partition the larger relation (R) into buckets. Tuples in the first bucket, R_0 are used to probe the S_0 hash table constructed in the first phase. At the end of this phase, the results of $S_0 \bowtie R_0$ are written to disk.

3. For all the remaining buckets, retrieve each bucket, join tuples, and store results.

5.1.1 Phase I: Hash Partition S-relation

Table 3 describes the hash partition phase. In the first two steps, $\mathbf{T}_i^S K^S$ tuples are processed. In step (3), $\mathbf{T}_i^S K^S \mathbf{S}_i^S \hat{\alpha}^S$ tuples remain after local selection. In step (4) $\mathbf{R}_i^S$ is the fraction of S tuples assigned to node i. It follows that $1 - \mathbf{R}_i^S$ of the tuples on node i must be moved.

In step (5), the number of tuples received at a node is the sum of tuples sent from all other nodes. The number of tuples sent from node j to node i ($i \neq j$) is $\mathbf{T}_j^S K^S \mathbf{S}_j^S \hat{\alpha}^S \mathbf{R}_i^S$. After redistribution (step 6), the number of tuples at node i is simply $\mathbf{R}_i^S \hat{\alpha}^S N K^S$. If B is the number of hash buckets, $1/B$ of the S-tuples are retained in memory, and the rest are saved on disk (step 6).

5.1.2 Phase II: Hash Partition R-relation

In phase II, The R relation (the larger relation) is partitioned, and tuples in the the first bucket are joined. The processing and analysis of the first six steps is identical to that described for phase I. The only difference between Table 3 and the first 6 steps in Table 4 is in the superscripts that designate relations.

Steps (7) to (9) of Table 4 show the costs required to process the hash bucket held in memory. There are $\hat{\alpha}^R N K^R$ R-tuples in the entire system; $\mathbf{R}_i^R \hat{\alpha}^R N K^R$ tuples are assigned to node i. Since only one bucket is joined in phase II, only $1/B$ of these tuples are processed. In step (7), the in-memory hash table of S-tuples that was built during phase I is probed with the R-tuples. In steps (8) and (9), $1/B$ of the join output is formed and written to disk. The cardinality of the join is the product of the size of the inputs ($\mathbf{R}_i^S \hat{\alpha}^S N K^S$ for the S-relation and $\mathbf{R}_i^R \hat{\alpha}^R N K^R$ for the S-relation), and the join selectivity. Recalling section 3.2.4, the *local* join selectivity for the partition at node i is $\mathbf{J}_i^{RS} N \hat{\rho}^{RS}$.

5.1.3 Phase III: Join Phase

During phase III, buckets that were written to disk during the first two phases are retrieved and joined. Since one bucket is processed in phases I and II, $(B-1)/B$ of all tuples are processed in phase III. Thus, the cost expressions in Table 5 all contain a $(B-1)/B$ term. Table 5 describes the costs of the join phase when there is no bucket overflow. Overflow is discussed in the next section.

In step (1), S-tuples are retrieved from disk. There are $\hat{\alpha}^S N K^S$ tuples on all nodes, with $\mathbf{R}_i^S \hat{\alpha}^S N K^S$ tuples on node i. Note that the cost is the same as step (6) in Table 3 (storing the buckets). In step (2), the S-tuples are loaded into a hash table. There are $\mathbf{R}_i^R \hat{\alpha}^R N K^R$ R-tuples on node i. In step (3), R-tuples are retrieved from disk; this cost is the same as step (6) in Table 4 (storing R-buckets). Step (4) shows the costs of probing the hash table with R-tuples. Tuples are joined in step (5). Again, the cardinality of the join output is the product of the input cardinalities ($\mathbf{S}_i^S \hat{\alpha}^S N K^S$ and $\mathbf{S}_i^R \hat{\alpha}^R N K^R$) and the join selectivity ($\mathbf{J}_i^{RS} N \hat{\rho}^{RS}$). Step (6) shows the cost of saving the join

step	number of tuples processed (n)	resource used	processing time
1. retrieve	$\mathbf{T}_i^S K^S$	disk	$\Theta(n, D, L^S)\, t_{disk}$
2. filter	$\mathbf{T}_i^S K^S$	CPU	nt_{filter}
3. hash	$\mathbf{T}_i^S K^S \mathbf{S}_i^S \hat{\alpha}^S$	CPU	nt_{hash}
4. send	$\mathbf{T}_i^S K^S \mathbf{S}_i^S \hat{\alpha}^S (1 - \mathbf{R}_i^S)$	comm	$\Theta(n, m, L^S)\, t_{send}$
5. receive	$\mathbf{R}_i^S \sum_j^{j \neq i} \mathbf{T}_j^S K^S \mathbf{S}_j^S \hat{\alpha}^S$	comm.	$\Theta(n, m, L^S)\, t_{recv}$
6. store	$\frac{B-1}{B} \mathbf{R}_i^S N \hat{\alpha}^S K^S$	disk	$\Theta(n, D, L^S)\, t_{disk}$

Table 3: phase I of Hybrid Hash: partition S-relation

step	number of tuples processed (n)	resource used	processing time
1 to 6	similar to phase I		
7. probe	$\frac{1}{B} \mathbf{R}_i^R N \hat{\alpha}^R K^R$	CPU	nt_{probe}
8. join	$\frac{1}{B} \mathbf{R}_i^R N \hat{\alpha}^R K^S \mathbf{R}_i^S N \hat{\alpha}^S K^R \mathbf{J}_i^{RS} N \hat{\rho}^{RS}$	CPU	nt_{join}
9. save	$\frac{1}{B} \mathbf{R}_i^R N \hat{\alpha}^R K^S \mathbf{R}_i^S N \hat{\alpha}^S K^R \mathbf{J}_i^{RS} N \hat{\rho}^{RS}$	disk	$\Theta(n, D, L^{join})\, t_{disk}$

Table 4: Phase II of Hybrid Hash: partition R-relation

step	number of tuples processed (n)	resource used	processing time
1. fetch S	$\frac{B-1}{B} \mathbf{R}_i^S N \hat{\alpha}^S K^S$	disk	$\Theta(n, D, L^S)\, t_{disk}$
2. hash S	$\frac{B-1}{B} \mathbf{R}_i^S N \hat{\alpha}^S K^S$	CPU	nt_{hash}
3. fetch R	$\frac{B-1}{B} \mathbf{R}_i^R N \hat{\alpha}^R K^R$	disk	$\Theta(n, D, L^R)\, t_{disk}$
4. probe R	$\frac{B-1}{B} \mathbf{R}_i^R N \hat{\alpha}^R K^R$	CPU	nt_{probe}
5. join	$\frac{B-1}{B} \mathbf{R}_i^R N \hat{\alpha}^R K^R \mathbf{R}_i^S N \hat{\alpha}^S K^S \mathbf{J}_i^{RS} N \hat{\rho}^{RS}$	CPU	nt_{join}
6. store	$\frac{B-1}{B} \mathbf{R}_i^R N \hat{\alpha}^R K^R \mathbf{R}_i^S N \hat{\alpha}^S K^S \mathbf{J}_i^{RS} N \hat{\rho}^{RS}$	disk	$\Theta(n, D, L^{join})\, t_{disk}$

Table 5: Phase III of Hybrid Hash: perform join

output. ($L^{join} \equiv L^R + L^S - L^{key}$).

5.1.4 Bucket Overflow in GAMMA Hybrid Hash

During the hash partition phases, tuples are read from disk, filtered, hashed and possibly dispatched to another node. No memory is required except buffers for disk and communications I/O. Because memory requirements are limited and not influenced by data cardinality or distribution, it may be assumed that overflow does not occur.

The hash join phase is different because all S-tuples must be held in an in-memory hash table in order to perform the join. Thus, the algorithm is vulnerable to overflow at this point.

As described in [Schneider 89], the GAMMA Hybrid Hash parallel join algorithm employs recursive hashing to resolve hash bucket overflow. When the hash table fills all available memory before the current bucket is completely processed, the initial hash function h_0 is replaced with a new hash function h_1. The new hash function is applied to the tuples in the hash table, with the result that a fraction p of the tuples are moved to an overflow file. Thereafter, any remaining tuples are hashed with h_1.

If the hash table overflows again, a new hash function h_2 is applied to remove $2p$ of the tuples. In general, the hash function h_G removes Gp of the tuples from the hash table. (the published descriptions of the GAMMA Hybrid Hash do not cover the case where $G > 1/p$). Let F be the number of tuples in the bucket. When all tuples in the bucket have been read, the hash table contains $(1 - Gp)F$ tuples. If the capacity of the hash table is C ($= UM/L^S$), then:

$$C > (1 - Gp)F \implies G = \left\lceil \frac{\text{F-C}}{Fp} \right\rceil \qquad (9)$$

The initial iteration of the recursive hash algorithm requires GC hash calculations, and $\Theta(GpF, D, L^S)$ disk writes to store S-tuples to the overflow file. Once processing of the S tuples is complete, the last hash function h_G is used to determine which R tuples are written to the overflow file. We assume that the hash functions partition the R file in the same proportions as the S-file. Let:

$$Y = \frac{\mathbf{R}_i^R N \hat{\alpha}^R K^R}{\mathbf{R}_i^S N \hat{\alpha}^S K^S} \qquad (10)$$

Then $\Theta(YGpF, D, L^R)$ disk writes are needed to write R-tuples to the overflow file. No extra hash calculations are required

After the join output tuples for the current hash table are written to disk, the overflow files are read into memory and a new hash table is constructed. If overflow occurs again, the above procedure is applied recursively. Overflow can occur in both the second and third phases of Hybrid hash. We assume that the second phase always processes $1/B$ of all tuples. In the third phase, it is assumed that all the overflowing tuples are concentrated in one bucket. F, the number of tuples in that bucket, is:

$$F = \frac{B-1}{B} \mathbf{R}_i^S \hat{\alpha}^S N K^S - (B-2)C \qquad (11)$$

5.2 Analysis of the Scheduling Hash Algorithm

The Scheduling Hash algorithm has four phases:

1. scan both relations and determine the number of tuples in each hash partition.
2. collect partition size information at one node and load balance the assignment of partitions to processors. Return the assignments to all processors.
3. Read tuples again and assemble each partitions at its join site.
4. perform a hash join of each partition.

We define B_1 and B_2, the number of coarse and fine hash partitions respectively, as follows:

$$B_1 \equiv N \quad B_2 \equiv \frac{\hat{\alpha}^S K^S}{UM} \qquad (12)$$

5.2.1 Phase I: Scan Phase

The scan phase collects information on how many tuples hash into each partition. Table 6 shows the processing steps for this phase. The S-relation is processed first: tuples are read from disk (step 1), local selection is applied (step 2), both coarse and fine hash keys are computed (step 3), and the selected tuples are written back to disk (step 4). R-tuples are processed in a similar manner in steps (5) to (8).

5.2.2 Phase II: Scheduling

The scheduling phase begins by counting the number of tuples on each node that hash into each hash partition. These counts are then collected at a single site. If the message traffic follows a logical binary tree pattern, results from all nodes can be collected in $\log_2 N$ message hops. Since the CPU cost of adding the counts from two nodes is negligible, the cost of collecting partition counts can be estimated as:

$$\log_2 N \Theta(B_1 B_2, m, L^{count})(t_{send} + t_{recv}) \qquad (13)$$

Where L^{count} is the number of bytes required to store the tuple counts for a single hash partition.

Next, the actual scheduling is performed. This requires subdividing the join into a number of tasks and assigning tasks to nodes. According to [Wolf 90], the time complexity of the scheduling is dominated by several sorts of these tasks. If we assume that the number of tasks is on the order of the number of partitions, then the CPU cost of the scheduling phase can be estimated as:

$$B_1 B_2 \left(\log_2 B_1 B_2 + \log_2 N \right) t_{schedule} \qquad (14)$$

Where $t_{schedule}$ is a proportionality constant for the scheduling algorithm(s).

Since it cannot be assumed that the interconnection network has a broadcast capability, transmitting the final assignment of tasks to the processors requires as much time as collecting the counts (see equation 13).

5.2.3 Phase III: Redistribution

Modeling redistribution skew in an algorithm that performs load balancing is problematic. If the load balancing were perfect, then RS and JPS would not exist. This is not a realistic assumption, nor is it plausible to assume the load balancing has no effect. Here, the effects of load balancing are approximated by "flattening" the skew vectors $\mathbf{T}_i$ and $\mathbf{J}_i^{RS}$ so that all elements of the vectors are between 90% and 110% of the mean value (for that vector). This approach captures

step	number of tuples processed (n)	resource used	processing time
1. read S	$\mathbf{T}_i^S K^S$	disk	$\Theta(n, D, L^S) t_{disk}$
2. filter S	$\mathbf{T}_i^S K^S$	CPU	nt_{filter}
3. hash S	$2\mathbf{T}_i^S K^S \mathbf{S}_i^S \hat{\alpha}^S$	CPU	nt_{hash}
4. store S	$\mathbf{T}_i^S K^S \mathbf{S}_i^S \hat{\alpha}^S$	disk	$\Theta(n, D, L^S) t_{disk}$
5. read R	$\mathbf{T}_i^R K^R$	disk	$\Theta(n, D, L^R) t_{disk}$
6. filter R	$\mathbf{T}_i^R K^R$	CPU	nt_{filter}
7. hash R	$2\mathbf{T}_i^R K^R \mathbf{S}_i^R \hat{\alpha}^R$	CPU	nt_{hash}
8. store R	$\mathbf{T}_i^R K^R \mathbf{S}_i^R \hat{\alpha}^R$	disk	$\Theta(n, D, L^R) t_{disk}$

Table 6: Phase I of Scheduling Hash: scan

step	number of tuples processed (n)	resource used	processing time
1. read S	$\mathbf{T}_i^S K^S \mathbf{S}_i^S \hat{\alpha}^S$	disk	$\Theta(n, D, L^S) t_{disk}$
2. send S	$\frac{N-1}{N}\mathbf{T}_i^S K^S \mathbf{S}_i^S \hat{\alpha}^S$	comm	$\Theta(n, m, L^S) t_{send}$
3. receive S	$\mathbf{R}_i^S \hat{\alpha}^S N K^S - \frac{1}{N}\mathbf{T}_i^S K^S \mathbf{S}_i^S \hat{\alpha}^S$	comm	$\Theta(n, m, L^S) t_{recv}$
4. store S	$\mathbf{R}_i^S \hat{\alpha}^S N K^S$	disk	$\Theta(n, D, L^S) t_{disk}$
5. read R	$\mathbf{T}_i^R K^R \mathbf{S}_i^R \hat{\alpha}^R$	disk	$\Theta(n, D, L^R) t_{disk}$
6. send R	$\frac{N-1}{N}\mathbf{T}_i^R K^R \mathbf{S}_i^R \hat{\alpha}^R$	comm	$\Theta(n, m, L^R) t_{send}$
7. receive R	$\mathbf{R}_i^R \hat{\alpha}^R N K^R - \frac{1}{N}\mathbf{T}_i^R K^R \mathbf{S}_i^R \hat{\alpha}^R$	comm	$\Theta(n, m, L^R) t_{recv}$
8. store R	$\mathbf{R}_i^R \hat{\alpha}^R N K^R$	disk	$\Theta(n, D, L^R) t_{disk}$

Table 7: Phase III of Scheduling Hash: redistribution

the assumption that load balancing in Scheduling Hash greatly reduces RS and JPS, but does not competely eliminate them.

The first step of the redistribution phase is to fetch S-tuples from disk (step 1), followed by transmission of data to join sites. In step (2), each partition is assigned to one of the N nodes with equal probability, so $\frac{N-1}{N}$ of the tuples stored on each node will require relocation. The number of tuples received at each node is the final population less the number of tuples that are already stored at their join site (step 3). In step (4), tuples are returned to disk. Processing of the R-relation in steps (5) to (8) is similar. Costs are summarized in Table 7

5.2.4 Phase IV: Join

The join phase is very similar to that of the Hybrid Hash. Not only are steps the same (see Table 5), but the expressions for the number of tuples are the same, except that there is no $(B-1)/B$ term. That is, the join phase of the Scheduling Hash processes all B buckets, while the join phase of the Hybrid Hash processes $B-1$ buckets. It is assumed that overflow does not occur in the Scheduling Hash.

6 Performance Results

A scalar model is used to generate skewed data distributions in these calculations. Skew is expressed as a scalar parameter Q: Let $\mathbf{X}$ be a skew vector, (one of $\mathbf{T}$, $\mathbf{S}$, $\mathbf{R}$, $\mathbf{J}$). Then

$$Q_{\mathbf{X}} \equiv \frac{N max_i(\mathbf{X}_i)}{\sum_i \mathbf{X}_i} \qquad (15)$$

In other words, one node has Q times as much data as the others. The remaining tuples are divided evenly among the other nodes.

Unless stated otherwise, skew is only in the inner relation (S, the smaller relation). Published measurements of partition skew are quite limited; there is some evidence[Walton 90] that Q values may be in the range of 2 to 3.

It should be emphasized that the vector relative partition model can represent arbitrary data distributions, and the above distribution was selected as a matter of convenience. Also, while in general several types of partition skew may occur simultaneously, each type will be considered in isolation to expose its effects on join performance.

The results presented here will concentrate on tuple placement skew (TPS), and redistribution skew (RS). In this test case, there is no selectivity skew because no tuples are eliminated by local selection. That is, selectivity is uniformly 100%. Join product skew (JPS) also has little effect on either algorithm due to the small cardinality of the join output. For the cases presented here, the computation is disk-bound.

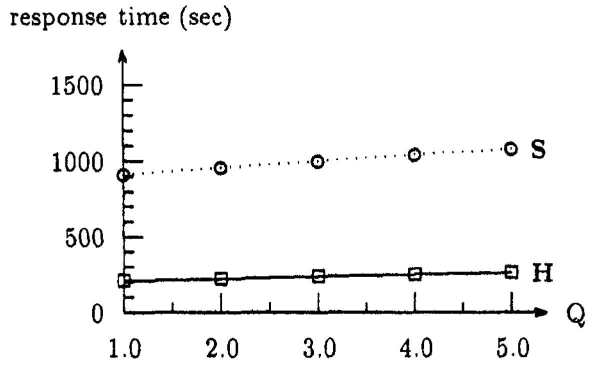

Figure 1: TPS comparison

Figure 1 compares the performance of Hybrid Hash and Scheduling Hash in the presence of TPS. For all values of the skew parameter, Hybrid Hash (line **H**) has better response time. Scheduling hash (line **S**) reads the data twice, once for the scheduling phase, and again for redistribution; Hybrid Hash reads the data once. As a result, Hybrid Hash requires less disk I/O even in the uniform case and is less sensitive to TPS (since fewer steps are affected by it).

The behavior of the two algorithms in face of redistribution skew, shown in Figure 2, is more interesting. Hybrid hash (line **HR**) has better performance at low Q values, while Scheduling Hash (line **SR**) is faster at higher Q values. In Hybrid Hash, RS can cause overflow, but the load balancing in Scheduling Hash prevents overflow. As redistribution skew becomes more pronounced, the overhead of reading tuples twice and performing load-balancing becomes less costly than processing bucket overflow. Also note the modest effect of join product skew (line **HJ**) on the response time of Hybrid Hash.

Figure 3 shows the effects of memory size on response time when redistribution skew (RS) is present in the

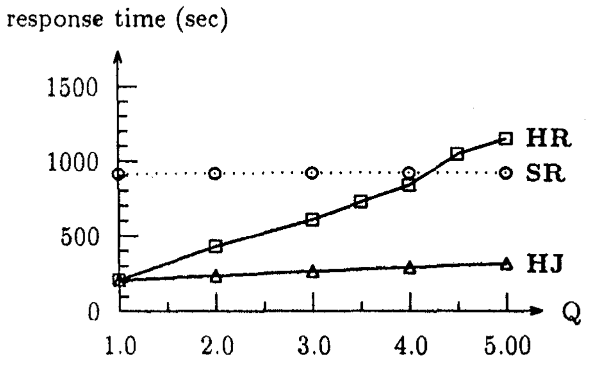

Figure 2: RS comparison

inner relation. Line **H1** shows the response time for Hybrid Hash with uniform data distribution ($Q = 1$). Lines **H2** and **H3** show response time for Hybrid Hash with Q values of 2 and 3 respectively. Line **S3** shows the response time for Scheduling Hash with $Q = 3$.

As line **H1** shows, even in the uniform case, increasing memory improves response time in Hybrid Hash. As available memory increases, a greater fraction of the relations can be allocated to the first bucket and joined during the second phase. This reduces the number of tuples that must be saved on disk and reread during the third phase. If there is enough memory, only one hash bucket is required, and phase III is eliminated.

Redistribution skew effects response time in two ways. First, it increases the number of tuples that are processed at some nodes. Furthermore, hash table overflow becomes more likely, leading to additional CPU and disk costs from the recursive hash procedure.

The interaction between the Hybrid Hash algorithm's memory sensitive behavior and the the recursive hash procedure explains the small increase in response time for the **H3** case when $M \approx 7$Mb. This is the point where all tuples can fit into one bucket (note the drop in response time for **H1**). However, in the **H3** case, this transition has the effect of concentrating all the overflowing tuples in a single bucket, instead of splitting them between two buckets. The greater excess of tuples leads to more iterations of the recursive hash algorithm, increasing the disk workload. That is, a greater portion of the tuples on the Q-node must be written to overflow files and read back several times before they are finally inserted in the hash table. In contrast, Scheduling Hash (line **S3**) is virtually unaffected by memory size.

The sensitivity of Hybrid Hash to memory size is also shown in Figure 4, which shows normalized speedup. The normalized speedup, $\hat{S}$, is defined as t_1/Nt_N, where t_1 is the response time for one node case and t_N is the response time for the N node case. An $\hat{S}$ value of 1 indicates linear speedup.

That is, doubling N halves the response time. Figure 4 covers the same cases as Figure 3. Because increasing N increases memory, superlinear speedups ($\hat{S} > 1$) occur. Increasing aggregate memory reduces disk I/O, which directly affects response time, as the computation is disk bound. These effects can be seen most clearly in the uniform case (**H1**), where the abrupt changes in $\hat{S}$ at $N = 12$ and $N = 23$ reflect changes in the number of buckets from 3 to 2 to 1.

For the skew cases, **H2** and **H3**, speedups are initially sublinear: by definition, skew (and overflow) cannot occur for $N = 1$, but the number of excess tuples is greatest when $N = 2$. As N increases, more memory becomes available and the effects of overflow are mitigated. Eventually, superlinear speedups are achieved.

Note the "staircasing" in **H2** and **H3**. As N increases, the amount of memory available per tuple increases, so that fewer iterations of the recursive hash are required.

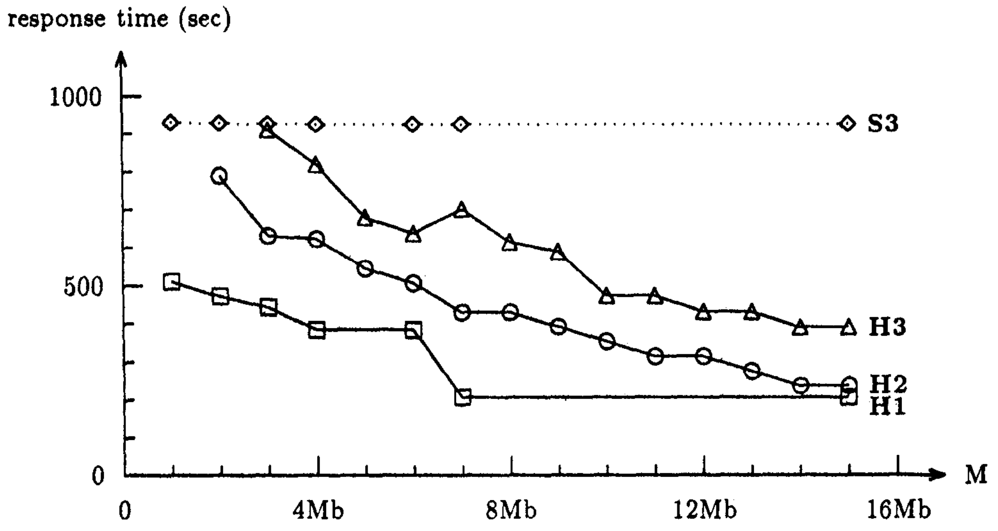

Figure 3: response time vs. memory capacity with RS

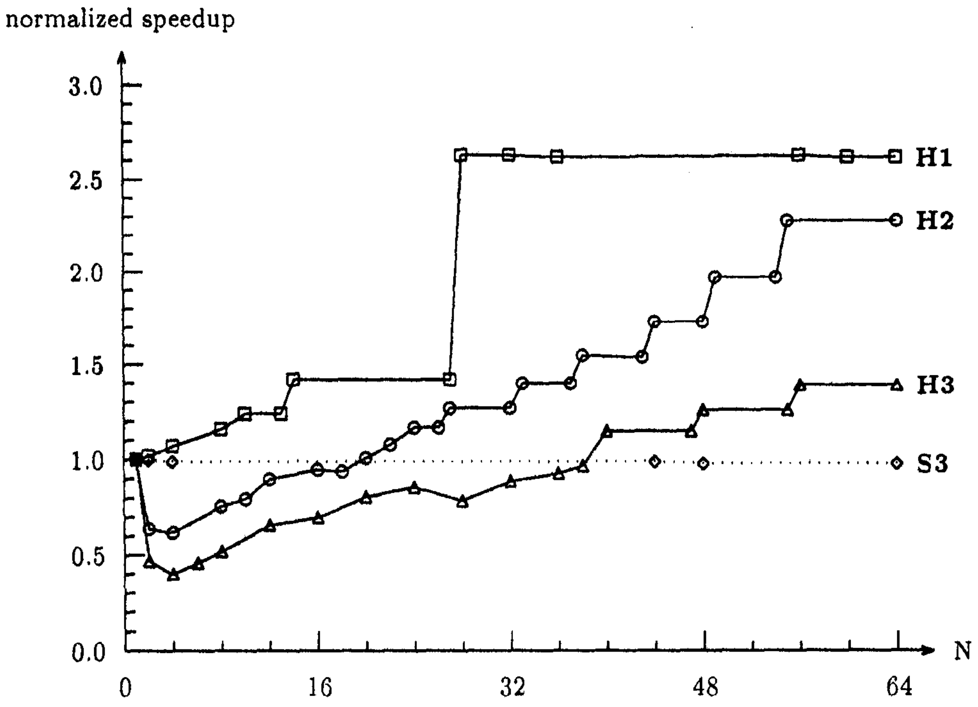

Figure 4: normalized speedup with redistribution skew

Because the fraction of tuples that can be written to overflow files can take on only a few distinct values, changes in normalized speedup are quantized. Additional memory will not change the number of tuples written to the overflow file until the increase is great enough to decrease the value of *Gp*. (see the discussion of recursive hash in section 5.1.4). Because the **H3** case has more severe redistribution skew, the increment of memory required to change response time (speedup) is larger than case **H2**. In contrast, Scheduling Hash is little affected by variations in memory and shows linear speedup.

The last two figures show examples of the differences between skew in the inner and outer relation. Figure 5 shows effects of TPS on Scheduling Hash. The interaction of skew in the inner and outer relations may be characterized as additive: any given step is affected by TPS in exactly one of the two relations. The greatest degradation in response time occurs when both relations have skew (line **SB**). Skew in the outer relation (line **SO**), which is 10 times larger than the inner relation, has nearly as much impact. Skew in the smaller inner relation (line **SI**) has a smaller effect. Finally, note that Hybrid Hash with skew in both relations (line **HB**) still has a better response time than any of the Scheduling Hash cases.

Figure 6 shows the effects of redistribution skew on Hybrid Hash. RS in the outer relation only (line **HO**) has little effect. It causes imbalances in the amount of communications and disk I/O during the second and third phases, but it does not cause overflow. Skew in the inner relation (line **HI**) has a much more pronounced effect, which indicates that overflow effects are responsible for most of the performance loss due to RS. If skew for the two relations are centered on different nodes, the effects are about the same as skew on the inner relation only. (line **HD**)

The worst case occurs when skew for both relations is centered on the same node (line **HB**). Unlike TPS, the interaction between skew in the inner and outer relations has a multiplicative nature. Redistribution skew in R amplifies the costs of overflow: skew in the outer relation causes the recursive algorithm to read and write an increased number of tuples each time it is applied.

For this particular query, Hybrid Hash seems to have better performance: it is faster in the uniform case, and with TPS. For RS, Hybrid Hash has a better performance except for for substantial skew.

However, a realistic evaluation would require much more information about the data and query characteristics, especially the extent of partition skew. Even without explicit load balancing, AVS need not result in partition skew. For example, there is some evidence that

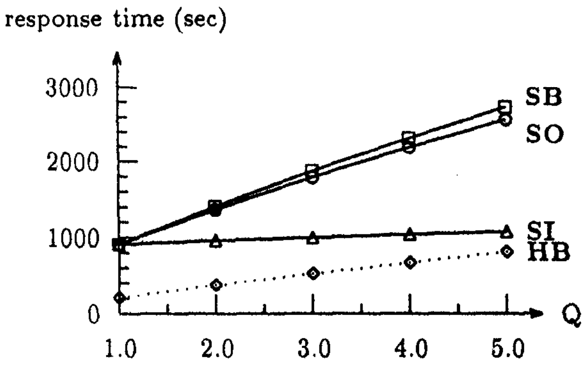

Figure 5: Scheduling Hash with TPS

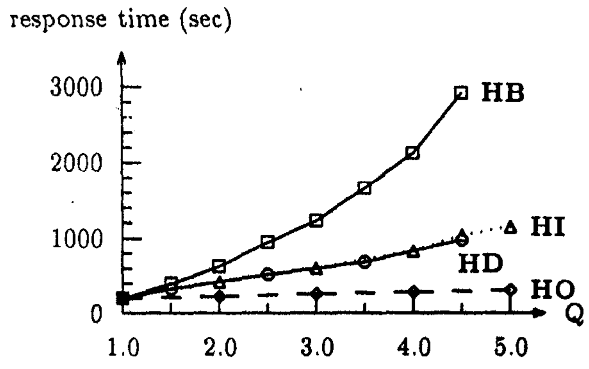

Figure 6: Hybrid Hash with RS

the redistribution scheme in GAMMA Hybrid Hash provides good load-balancing. Joins of normally distributed data, with very pronounced AVS, were reported in [Schneider 89,90]; but our approximate analysis suggests a Q values of only about 1.1 for RS.

7 Conclusions

The primary goals of this paper have been to explain shortcoming in previous models of data skew in parallel joins, and to show the advantages of the vector relative partition model. The results presented here illustrate some of the ways the model can be used to examine the effects of partition skew on join performance. Several conclusions can be reached from our work thus far:

- Skew is a heterogeneous phenomenon. In particular, there is a fundamental difference between intrinsic and partition skew. The relationship between the two is complex. A simple characterization of intrinsic skew is of little use in predicting partition skew.

- There are several types of partition skew, and their affects on algorithm performance differ. Also, sensitivity to partition skew varies between algorithms.

- In Hybrid Hash, overflow has a profound effect on response time, and accounts for most of the performance degradation due to redistribution skew.

The vector relative partition model is more comprehensive than scalar skew models. Its parameters have a simple operational definition, and it is a useful tool for evaluating algorithm performance. We anticipate using it to explore various aspects of data skew in parallel joins. In particular, we plan to examine the relationship between skew and scalability.

Acknowledgements Yasushi Kiyoki and Furman Haddix read early drafts of this paper and provided many useful comments.

References

Chaitanya K. Baru and Ophir Frieder. Database operations in a cube-connected multicomputer system. *IEEE Transactions on Computers*, C-38(6):920–927, June 1989.

Chaitanya K. Baru, Ophir Frieder, Dilip Dandlur, and Mark Segal. Join on a cube: analysis, simulation, and implementation. In *Database Machines and Knowledge Base Machines*, Kluwer Academic Publishers, 1987.

Haran Boral. Parallelism and data management. *3d Intl. Conf. on Data and Knowledge Bases*, Jerusalem (1988).

Stavros Christodoulakis. Estimating record selectivities. *Information Systems*, 8(2):105–115, 1983.

George Copeland, William Alexander, Ellen Baughter, and Tom Keller. Data placement in bubba. *1988 SIGMOD Proc.*, Chicago.

David J. DeWitt *et al.*, Gamma - a high performance backend database machine. *12th VLDB Proc.*. Kyoto (1986)

David J. DeWitt, S. Ghadeharizadeh, and Donovan Schneider. A performance analysis of the gamma database machine. In *1988 SIGMOD Proc.*, Chicago.

David J. DeWitt *et. al.*, The gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, March 1990.

David J. DeWitt, R. Katz, F. Olken, D. Shapiro, Michael Stonebraker, and D. Wood, Implementation techniques for main memory database systems. *1984 SIGMOD Proc.*, Boston.

Ophir Frieder, Multiprocessor algorithms for relational-database operations on hypercube systems. *IEEE Computer*, 23(11):13–28, November 1990.

Robert H. Gerber. *Dataflow Query Processing using Multiprocessor Hash-Partitioned Algorithms*. Technical Report 672, University of Wisconsin, October 1986.

Robert H. Gerber and David J. DeWitt. *The Impact of Hardware and Software Alternatives on the Performance of the GAMMA Database Machine*. Technical Report 708, University of Wisconsin, July 1987.

R.-C. Hu and R. Muntz. *Removing Skew Effect in Join Operations on Parallel Processors*. Technical Report CSD-890027, UCLA, June 1989.

M. Kitsuregawa, M. Nakano, and T. Moto-Oka. Application of hash to database machine and its architecture. *New Generation Computing*, 1(1), 1983.

M. Seetha Lakshimi and Philip S. Yu. Effect of skew on join performances in parallel architecture. *Symp. on Databases in Parallel and Distributed Systems Proc.*, Austin (1988)

M. Seetha Lakshimi and Philip S. Yu, Limiting factors of join performance on parallel processors. *Conference on Data Engineering*, Los Angeles (1989).

C. A. Lynch, Selectivity estimation and query optimization in large databases with highly skewed distributions of column values. *14th VLDB Proc.*, Los Angeles (1988).

Anthony Y. Montgomery, Daryl J. D'Souza, and S. B. Lee. The cost of relational algebraic operations on skewed data: estimates and experiments. In *Information Processing 83*, Elsiver Science Publishers, Amsterdam, 1983.

Edward Omiecinski and Eileen Tien Liu. *The Adaptive-Hash Join Algorithm for a Hypercube Multicomputer*. Technical Report GIT-ICS-89/48, Georgia Institute of Technology, December 1989.

James P. Richardson, Hongjun Lu, and Krishna Mikkilineni. Design and evaluation of parallel pipelined join algorithms. *1987 SIGMOD Proc.*, San Francisco.

Donovan A. Schneider. Complex query processing in multiprocessor database machines. *16th VLDB Proc.*, 1990.

Donovan A. Schneider and David J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. *1989 SIGMOD Proc.*, Portland, Oregon.

Michael Stonebraker, The case for shared nothing. *Database Engineering*, 9(1), March 1986.

DBC/1012 Database Computer Concepts and Facilities. Teradata Corporation, 1983.

Christopher B. Walton, Matt L. Pinsonneaut, and Furman Haddix, *Measurements of Data Skew in Two Databases*. Technical Report TR-90-32, University of Texas at Austin, October 1990.

Joel L. Wolf, Daniel M. Dias, and Philip S. Yu, *An Effective Algorithm for Parallelizing Hash Joins in the Presence of Data Skew*. Research Report RC 15510, IBM Watson Research Center, February 1990.

Joel L. Wolf, Daniel M. Dias, and Philip S. Yu. An effective algorithm for parallelizing sort merge joins in the presence of data skew. *Databases in Parallel and Dist. Systems*, Los Almitos, CA, (1990).