Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer (SDC)

Masaru Kitsuregawa
Institute of Industrial Science,
University of Tokyo
7-22-1 Roppongi, Minato-ku,
Tokyo 106, Japan

Yasushi Ogawa
Research and Development Center,
RICOH Co., Ltd.
16-1 Shinei-cho, Kohoku-ku,
Yokohama 223, Japan

Abstract

The *Super Database Computer* (SDC) is a high-performance relational database server for a join-intensive environment under development at University of Tokyo. SDC is designed to execute a join in a highly parallel way. Compared to other join algorithms, a hash-based algorithm is quite efficient and easily parallelized, and has been employed by many database machines. However, in the presence of data skew, it's hard to distribute load equally among processing modules (PMs) by *statically* allocating buckets to PMs, as in the conventional parallelizing strategy. Thus, performance is severly degraded.

In this paper, we propose a new parallel hash join method, the *bucket spreading strategy*, which is robust for data skew. During partitioning relations, each bucket is again divided into fragments of the same size and these fragments are temporarily placed on PMs one by one. Then each bucket is *dynamically* allocated to a PM which actually carries out the join of the bucket, and all fragments of the bucket are collected in the corresponding PM. In this way, the bucket spreading strategy evenly distributes the load among the PMs and parallelism is always fully exploited. The architecture of SDC is designed to support the bucket spreading strategy; a mechanism which distributes the buckets flatly among the PMs is embedded in the hardware of the interconnection network. Simulation results confirm that the bucket spreading strategy is robust for data skew and attains very good scalability.

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

1 Introduction

Once relational database systems became widely used, considerable research has been focused on parallel relational database machines [Kit83,DeW86,Ter88, Tan88,Bra89]. The *Super Database Computer* (SDC) is a high-performance relational database server for a join-intensive environment which is under development based on our previous researches: the database machine GRACE[Kit83,Kit84], the Functional Disk System[Kit87,Kit89], and the high-speed hardware sorter[Kit87b,Kit89b]. SDC is designed to execute a join efficiently by introducing highly parallel processing. Since a join is one of the most expensive and complicated operations among the relational operations, many algorithms have been proposed. Hash-based algorithms, such as GRACE [Kit83], Simple or Hybrid [DeW84], partition a relation into a number of clusters called *buckets*. Because a join operation is divided into small joins of buckets, hash-based algorithms perform much better than other algorithms such as nest-loop or sort-merge [Sha86]. Following the trend of applying parallelism to database processing, many have proposed efficient parallel join methods [Kit83,Bra84,Val84,DeW85,Omi89]. Since each small join of buckets can be carried out independently, hash-based join algorithms are suitable for parallel processing [Kit83]. In [Kit83], the author classified the possible database machine architecture for hash based relational processing. So far, a straightforward strategy is used to make a hash join parallel in such as GAMMA [Ger86] and Teradata DBC/1012 [Ter88]. The processing module (PM) for each bucket is determined *statically* based on its bucket ID before partitioning relations. Because relations are *horizontally partitioned* across the PMs, relations are divided in parallel and each bucket is gathered into the corresponding PM through the interconnection network. Each PM then conducts the joins of all allocated buckets. This method is referred to as a *bucket converging strategy* [Kit83].

In parallel systems, data skew deteriorates the performance severely without a load balancing mech-

anism. Database processing, including join, is no exception, and conventional parallelized hash-based methods face performance degradation [Kit83,Lak88, Lak89]. When using the bucket converging strategy, performance is degraded as parallelism is decreased due to a heavy load unbalance which results from a large portion of the processing load being concentrated in only a few PMs. This is because the data is skewed by static bucket allocation before partitioning.

In this paper, we propose an alternative parallel hash join method to solve the problem caused by data skew. If one allocates buckets to PMs *dynamically* based on their bucket size, the PM bucket volume can be maintained equally for all PMs. Thus, the load of a join is evenly distributed across PMs, and consequently a join operation can be processed efficiently despite an unbalanced data distribution. To allocate buckets to PMs dynamically, each bucket is divided into nearly equal fragments, which are temporarily placed on different PMs. In order to efficiently collect the fragments during a subsequent bucket join, the PMs gather buckets simultaneously by using a rotational scheduling. Because a bucket is *spread* out among the PMs we call this method a *bucket spreading strategy* [Kit83].

The architecture of SDC is designed to support efficient execution of a bucket spreading parallel hash join. Compared to the bucket converging strategy, the bucket spreading strategy requires additional control to distribute buckets flatly among the PMs. To decrease the effect of overhead which could slow down performance considerably, the interconnection network of SDC embeds the hardware function which distributes the buckets evenly across the PMs. Therefore the PMs are not subjected to unnecessary additional processing. This highly-functional network is also addressed in this paper.

In the next section, we briefly explain SDC. Section 3 describes the conventional bucket converging strategy and restates its problems caused by data skew. In Section 4, we show that dynamic bucket allocation solves the problems. Then, in Section 5, we propose the bucket spreading strategy which actualizes the dynamic bucket allocation. Section 6 explains the architecture support of the bucket spreading strategy in SDC. Performance evaluation of the bucket spreading strategy is presented in Section 7. Finally, we present our concluding remarks in Section 8.

2 Architecture of SDC

This section is a brief explanation of the architecture of the *Super Database Computer* (SDC). SDC is a parallel relational database server under development at University of Tokyo based on our earlier researches: the database machine GRACE[Kit83, Kit84], the Functional Disk System[Kit87,Kit89], and the high-speed hardware sorter[Kit87b,Kit89b]. The basic architecture of SDC is *shared-nothing* [Sto86], and relations are *horizontally partitioned* [Rie78]. SDC's features are summarized as follows (for details, please refer to [Hir90]):

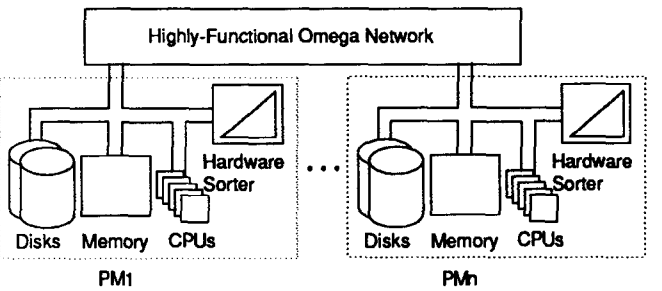

Figure 1: Architecture of Super Database Computer

Hybrid Parallel Architecture

Figure 1 illustrates the architecture of SDC. SDC consists of several processing modules (PMs) connected to each other through the message-passing interconnection network. In order to realize *on the fly* processing of the data stream from disk, each PM itself is designed as a tightly coupled multiprocessor system which has strong computational power. Thus the architecture is hybrid of tightly coupled multiprocessor (shared everything) and the message-passing architecture (shared nothing).

Integration of the Processing and Storage Systems

Each PM consists of a processing system (multiprocessor) and a storage system (hard disks), which is of the same configuration as the Functional Disk System (FDS) [Kit87,Kit89]. FDS was proposed to provide an optimal I/O environment for data intensive applications, to solve the impedance mismatch between the operating system and the DBMS, and to provide hardware support for relational processing.

Highly-functional Interconnection Network

SDC employs a highly-functional omega network for the interconnection network. Flat bucket distribution, required by the bucket spreading strategy, is actualized by the network hardware with-

out creating an additional load for the PMs. The network is detailed in Section 6.

High-Speed Hardware Sorter
Each PM has a hardware sorter composed of LSI sort chips [Kit87b,Kit89b][1]. Because a sort is performed in linear time by the sorter, high-speed database processing can be realized. The memory for the sorter is *bimodal*; the sort memory can be accessed as ordinary RAM by the processors. Thus the total memory space of PM is shadowed on the sort memory.

Separation of Data and Control Passes
When the number of PMs is large, much more control information, such as synchronization both between PMs and between processors in a PM, becomes necessary. So as not to obscure the data stream, control passes are completely separate from data passes. SDC has two interconnection networks between the PMs, and there are two internal busses in each PM.

All parameters, such as the bandwidth of the internal bus in PM, the aggregate performance of processors and the performance of the sorter, are designed to match the data transfer rate from the disk. That is, all relational operations can be performed keeping up the data stream from the disk. Furthermore, data processing and data transfer is overlapped as much as possible.

3 Problems with Conventional Parallel Hash Join Methods

In this section, we explain conventional parallel hash join methods and clarify their problems of data skew. The two source relations of a join will be referred to as R and S, where S is larger than R. Each relation is assumed to be larger than the aggregate memory capacity, which is the sum of the memory capacity across all PMs.

In the following discussion, the GRACE hash algorithm [Kit83] is assumed to be a hash join algorithm to simplify the explanation, although the following study can be applied to other algorithms including the *popular* Hybrid hash algorithm [DeW84] which is a combination of the GRACE hash and the Simple hash algorithms. The GRACE hash algorithm consists of two distinct stages: a *split phase* and a *join phase*. During the split phase, relations R and S are partitioned into clusters called *buckets*. Joins of corresponding buckets, i.e., buckets from the two relations which share the same bucket ID, are performed in the join phase.

In hash-based join algorithms, the size of each bucket should be smaller than the memory. However, data skew occasionally generates *bucket overflow*, in which buckets are larger than the memory. Performance is diminished, because bucket overflow requires extra I/O to repartition buckets to be smaller than the memory. The performance diminish can be avoided by *bucket size tuning*; the number of buckets in the split phase is set to a large number so as not to generate bucket overflow, and then several small buckets are combined into one bucket to fit the memory capacity [Kit83,Kit89c].

3.1 Bucket Converging Strategy

Because tuples of a relation in one bucket are never joined with tuples of the other relation in another bucket, corresponding buckets can be handled independently. Thus, the hash join method is suitable for parallel processing [Kit83,DeW85]. In the architecture considered here, relations are partitioned into buckets in parallel because relations are horizontally partitioned. Joins of buckets are also executed in parallel. In the conventional parallel hash join method, each bucket is *statically* allocated to a PM based on its bucket ID at the beginning of the join operation [Ger86,Sch89]. During partitioning, every tuple in a bucket is collected to the corresponding PM through the interconnection network, and each bucket is converged in a single PM. Figure 2 illustrates the case when the number of PMs is 4.

A GRACE hash join is executed straightforwardly in parallel as described below. In the split phase, R is first partitioned in parallel. Each PM independently divides a portion of the relation stored therein. If a tuple belongs to buckets allocated to the PM, it is written back to the PM's local disk. Otherwise, it is transferred to the PM corresponding to its bucket ID through the interconnection network, and stored in that disk. When all of the PMs have finished partitioning, the whole relation has been partitioned. The partitioning of S is carried out in the same way. Bucket size is tuned after partitioning the relations. Each PM can tune the size of the allocated buckets independently because they are already converged there.

In the join phase, each PM performs small joins of the allocated buckets. Because the PM already stores the allocated buckets in its local disk, there is no need

[1]Currently this hardware sorter is made into commercial product of Mitsubishi Electric Corp. [Mit89], attached to office processor.

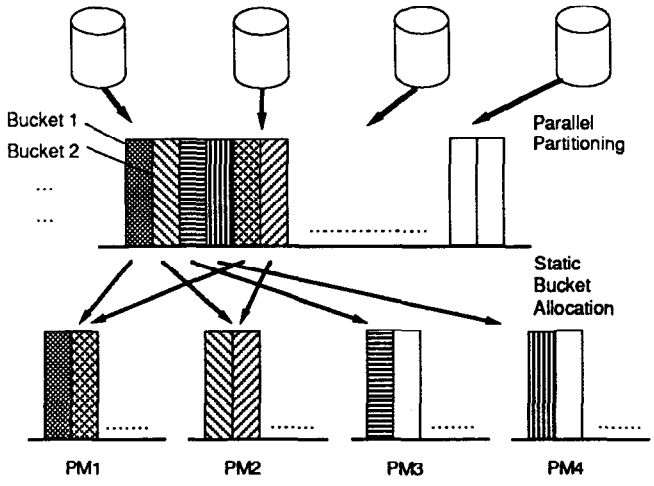

Figure 2: Parallel Hash Join Using Bucket Converging Strategy

to communicate with other PMs. Thus each PM conducts its join independent of other PMs in the same way as a single PM system. Until all allocated buckets are processed a PM repeats the following procedure; the PM reads a pair of the corresponding buckets from its local disk and joins them. The whole join operation is finished when all the PMs have finished their joins. Because each bucket is stored in a single PM after the partition, this method is named *bucket converging strategy* [Kit83].

3.2 Problems of Data Skew

In parallel systems, data skew degrades the performance severely unless there is a load balancing mechanism. A join operation also faces the performance degradation by data skew [Kit83,Lak88,Lak89]. Since the source relations usually result from earlier operations such as selection or projection, data distribution is hard to predict. So buckets tend to vary in size (*bucket size fluctuation*).

Although hash-based join algorithms are easily made parallel using the bucket converging strategy, performance is greatly slowed down by data skew. There are two possible major reasons for the poor performance. The first is bucket overflow, and the other is a skew of total PM bucket volume. A skew of total PM bucket volume, which leads to load unbalancing as illustrated in Figure 3, means an imbalance in the total volume of data among the PMs. Bucket overflow is almost avoided by employing bucket size tuning in the same way as a single processor system [2]. However, a skew of the total PM bucket volume cannot be avoided by bucket size tuning due to the fact that the static bucket allocation creates the skew. The skew reduces the degree of parallelism, resulting in severe degradation of the total processing performance.

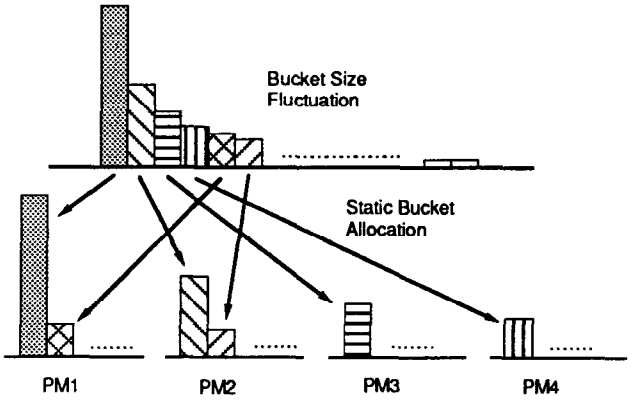

Figure 3: Load Unbalancing Caused by Data Skew

Data skew further degrades performance in the following way. During the split phase, a PM whose total bucket volume is very large has to gather a lot of tuples from other PMs. That is, because the source relations are partitioned to buckets in parallel by the PMs, every PM has to gather tuples which come from the other PMs and belong to the bucket allocated to that PM. When a bucket is very large, tuples from the bucket are transferred concentratedly to the corresponding PM, resulting in network *hot spot*. Hot spots further degrade performance. In addition, data skew causes problem in disk space. When joining large relations using hash-based algorithms, buckets are written once to disk. When bucket sizes are highly unbalanced, some PMs, which are allocated huge buckets, cannot hold the buckets in their local disks even if the total disk space over the PMs is enough to hold all of the buckets. Such a situation creates extra data transfer between PMs which decreases performance.

4 Dynamic Bucket Allocation

As described in the above section, static bucket allocation causes an imbalances on the PM bucket volume among the PMs and degrades performance. However, if a bucket is *dynamically* allocated to a PM based on its size, the PM bucket volume can be controlled so that it is equal for all PMs. As a result, performance is not decreased despite data skew.

[2] Although bucket size tuning is quite effective, it is impossible to guarantee the avoidance of bucket overflow. So we again consider the problem of bucket overflow in Section 5.5.

If bucket size is tuned before allocating buckets to PMs, bucket sizes become nearly uniform. In this case, fairly good performance is expected simply by allocating buckets in rotation; the ith bucket ($1 \leq i \leq B'$) is allocated to the $(((i-1) \bmod N)+1)$th PM and is processed in the $\lceil i/N \rceil$th cycle. Here B' is the number of buckets after bucket size tuning. It may seem that buckets are being allocated according to bucket ID since the scheduling method is cyclic just as in the bucket converging strategy. However, because the buckets are already tuned on size, the buckets are actually allocated according to their size. That is, by introducing bucket size tuning prior to allocation, the cyclic method balances the PM bucket volume, i.e. the processing load, among PMs. Figure 4 illustrates how buckets are dynamically allocated.

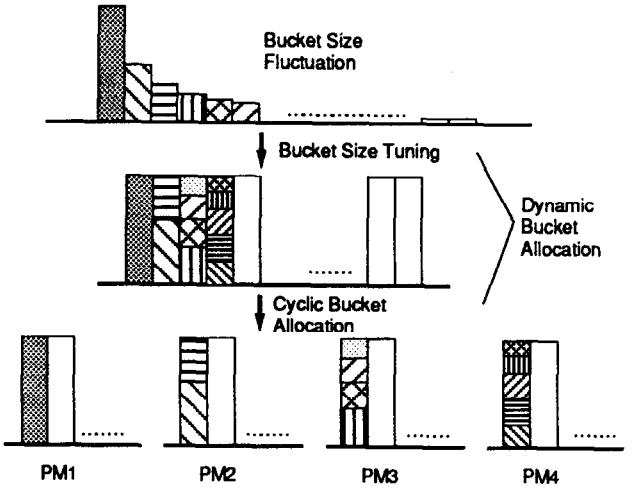

Figure 4: Load Balancing By Dynamic Bucket Allocation

Even after bucket size tuning, however, the buckets vary slightly in size. To reduce the effect of this nonuniformity, we sort the buckets according to size. Then the ith bucket ($1 \leq i \leq B'$) is processed in the $\lceil i/N \rceil$th cycle by:

The $(((i-1) \bmod N)+1)$th PM
: when $\lceil i/N \rceil$ = odd
The $(N-((i-1) \bmod N))$th PM
: when $\lceil i/N \rceil$ = even

5 Bucket Spreading Strategy

In this section, we explain how dynamic bucket allocation can be realized using our proposed *bucket spreading strategy.*

5.1 Spreading Buckets among PMs

To dynamically allocate buckets we must consider the place storing each bucket during the partition of the relations. That is, in the bucket converging strategy, because a bucket was allocated to a PM before partitioning, which PM stores which tuple was determined simply from the tuple's bucket ID. However, when buckets are to be dynamically allocated, the destination of a tuple cannot be determined during the partition because which PM handles the tuple has not been determined yet. Where should the buckets be placed to store temporarily ?

We solve this problem by distributing each bucket among the PMs. A bucket is again partitioned into as many fragments as there are PMs and each fragment is placed into a PM. Figure 5 illustrates the spreading placement strategy. The portion of a bucket stored in each PM is referred to as a *subbucket.* We name this new method a *bucket spreading strategy* because each bucket is *spread* out among PMs.

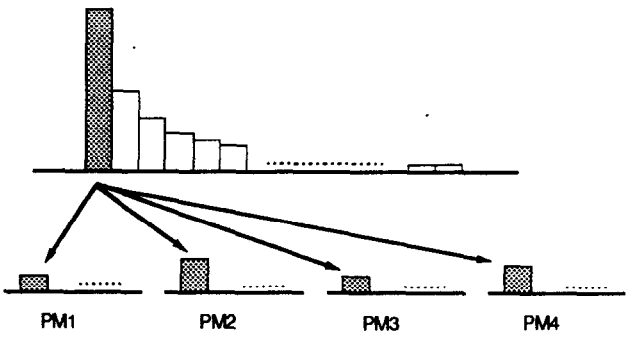

Figure 5: Spreading Placement of Bucket

5.2 Rotational Bucket Collection

In the bucket spreading strategy, the whole bucket is not placed in the PM which processes the bucket but spread out over the PMs. Thus, during the join phase, the PM has to gather the fragments of the bucket which are stored in other PMs. At that time, if all PMs gather buckets arbitrarily, an access conflict can occur and reduce performance. To avoid this conflict, some scheduling is necessary for bucket collection. Here we show how a simple rotational scheme can be used to avoid access conflict.

Collecting one bucket consists of N steps of N subbucket reads, where N is the number of PMs. N buckets are collected simultaneously during one cycle. The ith PM ($1 \leq i \leq N$) reads the subbucket of the ith bucket in the $(((i+j-2) \bmod N)+1)$th PM at the

jth step ($1 \leq j \leq N$). This kind of simultaneous connections between PMs is referred to as a *circular shift* and a series of small reads, a *bucket collection cycle*. It should be noted that, because $((i-1) \bmod N)+1 = i$ when $j = 1$, a PM reads a subbucket from its local disk at the first step. Figure 6 illustrates one bucket collection cycle. At each step, subbuckets surrounded by the bold line are read out from the disk.

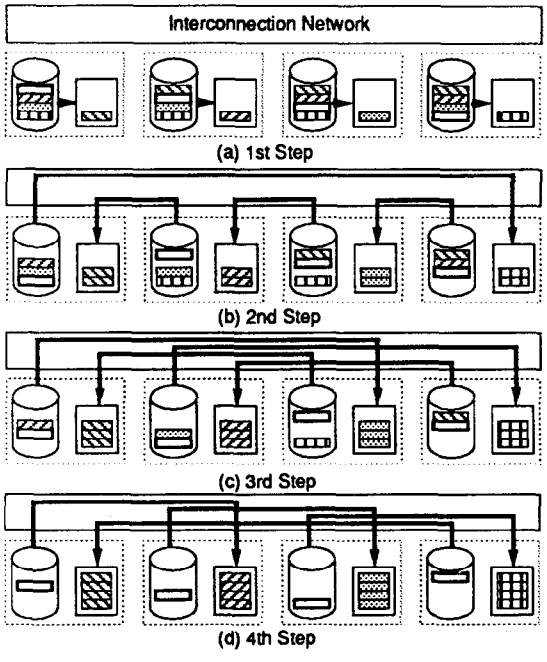

Figure 6: Bucket Collection Cycle

5.3 Flat Bucket Distribution

The time required for one bucket collection cycle is largely affected by a bucket distribution [3], since the time for a given step is determined by the size of the largest subbucket collected in that step. Although bucket size is nearly uniform after bucket size tuning, one bucket collection cycle takes much time if subbucket size of the buckets varies (Figure 7 (a)). Thus subbucket size should be uniform to minimize the time for one bucket collection cycle. That is, the buckets need to be distributed flatly among the PMs for efficient processing (Figure 7 (b)). We call this kind of bucket distribution a *flat bucket distribution*.

Since relations are horizontally partitioned, each PM divides a relation's portion stored in its local disk

[3]The distribution of subbuckets of one bucket among PMs is referred to as a *bucket distribution*. Note that it is different from the distribution of buckets of one relation.

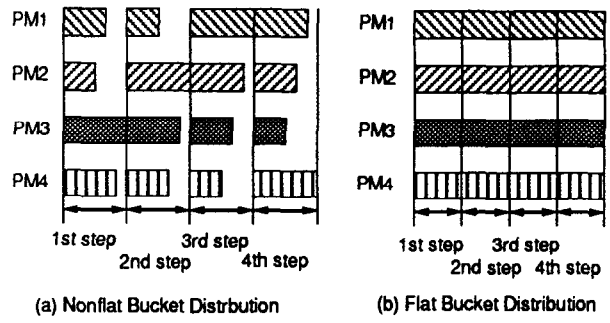

Figure 7: Effect of Bucket Distribution on a Collection Cycle

and writes buckets back to the disk. This implementation is very simple and does not use the interconnection network during the split phase. However, because data distribution normally differs in PMs as illustrated in Figure 8 (a), a flat bucket distribution cannot be realized using this simple implementation. To attain a flat bucket distribution (Figure 8 (b)), it is necessary to exchange tuples between PMs.

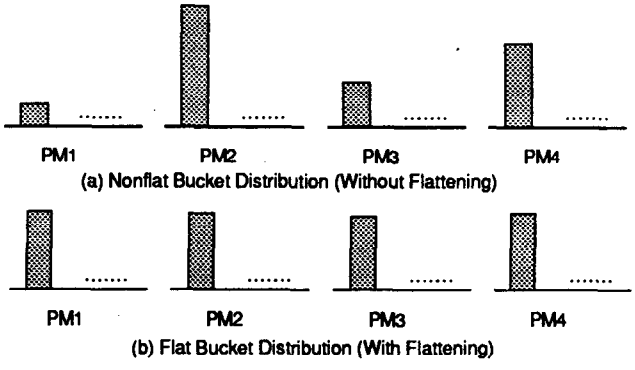

Figure 8: Flat Bucket Distribution

For a flat bucket distribution, basically each tuple is sent to the PM containing the smallest subbucket among all the subbuckets of its bucket. Thus, to determine the destination of a tuple, each PM has to keep track of data distribution on every PM. This requires a high volume of data transfer between PMs, resulting in considerable performance degradation. SDC avoids the problem by using a highly-functional omega network to interconnect the PMs, as explained in Section 6, so that no additional processing is required of the PMs.

5.4 Bucket Size Tuning

As described in Section 4, buckets can be allocated dynamically by tuning bucket size before allocation.

Compared to the bucket converging strategy where each PM independently tunes the size of the buckets assigned to it, the PMs must cooperate to tune bucket sizes because each bucket is spread out. Such bucket size tuning becomes a major task when the number of PMs is large. However, bucket size tuning can be carried out more simply. Since buckets are distributed flatly among the PMs, the distribution of subbuckets in each PM is representative of the distribution of buckets among all PMs. That means one master PM can decide how to tune the size of buckets based on its local distribution of subbuckets. Slave PMs can follow the bucket size tuning determined by the master PM.

5.5 Other Advantages

Here, we mention two other advantages of the bucket spreading strategy: repartitioning of overflowed buckets and disk space conservation.

Even introducing bucket size tuning, some buckets may exceed the available memory space. An overflowed bucket has to be repartitioned into small fragments so that each fits memory. Thus a bucket which overflows requires extra I/O equal to twice the size of the bucket. In the bucket converging strategy, the extra I/O is concentrated in the PM holding it. Contrastly, since the overflowed bucket is spread out among PMs, the additional I/O is also distributed on them. Thus the efficiency of the bucket spreading strategy is not so severely affected by bucket overflow as the bucket converging strategy.

Next, we consider the disk space problem. As mentioned in Section 3.2, if data is skewed, some PMs could lack enough disk space to hold the buckets allocated to them in the bucket converging strategy. However, in the bucket spreading strategy, because buckets are distributed equally among all PMs, the disk space required of each PM to hold all allocated subbuckets is equal across all PMs. In other words, the disk space required to hold buckets is also evenly divided among PMs, so disk problem rarely occurs and the disk space allocation can be simplified.

5.6 Parallel GRACE Hash Join Using Bucket Spreading Strategy

In the split phase, R and then S are partitioned in parallel. While a PM sends out all tuples to the interconnection network, the PM stores tuples, which are sent from other PMs through the network, on its local disk. When all PMs have completed their partitions, because the network itself has a function to distribute buckets flatly among the PMs, the buckets are spread out flatly without any destination control by PMs. After partitioning both relations, buckets are allocated dynamically to the appropriate PMs: buckets are allocated cyclicly after bucket size tuning.

During the join phase, each PM performs small joins of corresponding buckets. Buckets distributed among the PMs are gathered by employing rotational bucket collection as explained in Section 5.2. Although each PM performs joins of buckets independently in the bucket converging strategy, all the PMs are synchronized with bucket collection cycles in the bucket spreading strategy. PMs repeat this procedure until all the buckets have been joined.

6 Bucket Flattening Omega Network: Architectural Support of Bucket Spreading Strategy in SDC

In SDC, the architecture is designed to support efficient processing of a bucket spreading parallel hash join. If the interconnection network embeds a hardware function to distribute buckets flatly among the PMs, the PMs are not required to perform additional processing to flatten bucket distribution. Thus we've developed a bucket flatteing omega network for SDC.

The topology of the network is the same as that of an omega network, a kind of multi-stage network [Law75]. An $N \times N$ omega network[4] consists of identical $n = \log_2 N$ stages, and each stage has $N/2$ switching units (SUs). Figure 9 illustrates 8×8 omega network. Each SU is a 2×2 crossbar switch with either of two states, Crossed (Figure 10(a)) or Straight (Figure 10(b)). Although an omega network is a type of blocking network, it can allow circular shift connections without blocking required in the join phase of the bucket spreading strategy.

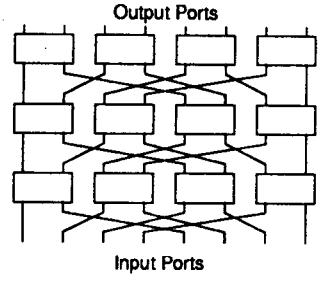

Figure 9: Omega Network

[4] N is usually a power of two.

Figure 10: Two States of Switching Unit

6.1 Distributed Control Method of Bucket Flattening Omega Network

There could be two bucket flattening methods: centralized and distributed. In the centralized control method, one central control module keeps track of bucket distribution among the PMs and determines the destinations of tuples. However, when the network is large, the control module becomes a bottle neck and throughput is reduced. Conversely, in the distributed control method, each SU is somewhat *intelligent* and decides its state autonomously based on local information. Thus it is applicable to larger networks. Therefore we adopt the distributed control and, in the following, explain the state determination algorithm of SU.

Each SU is equipped with the same number of counters as buckets. $D(X)$, the value of a counter for bucket X, keeps the difference of the number of tuples in bucket X output from the left output port (LOP) and the right output port (ROP) of the SU. All counters are initially set to 0. The value of a counter for a bucket is incremented if the tuple of the bucket is output from LOP, and decremented if from ROP. $D(X)$ represents the skew of the distribution of tuples in a bucket as seen from its SU. When $D(X) > 0$, the SU sends more tuples in bucket X from LOP than from ROP, and vice versa. X_{left} and X_{right} refer to buckets of tuples input to the left input port (LIP) and the right input port (RIP). The difference between counter values of X_{left} and X_{right}: $Dif = D(X_{left}) - D(X_{right})$ represents the relative skew of distributions of tuples in the bucket X_{left} and the bucket X_{right}. Therefore, to distribute a bucket flatly among PMs, the state of the SU is set to Crossed when $Dif > 0$, and Straight when $Dif < 0$. The state of the SU can be arbitrarily determined if $Dif = 0$.

An example of the determination of a state in a SU is shown in Figure 11. The SU receives tuples belonging to bucket m and n from LIP and RIP respectively (Figure 11(a)). Because the value Dif is positive, the SU is set to Crossed (Figure 11(b)). As a result, the tuple of bucket m leaves ROP and the counter for bucket m is decremented, while the tuple of bucket n leaves LOP and the counter for bucket n is incremented (Figure 11(c)).

It should be noted that, in the above algorithm, each SU always takes one of two possible states and never blocks tuples. That means there are neither *network conflicts* nor *hot spots*.

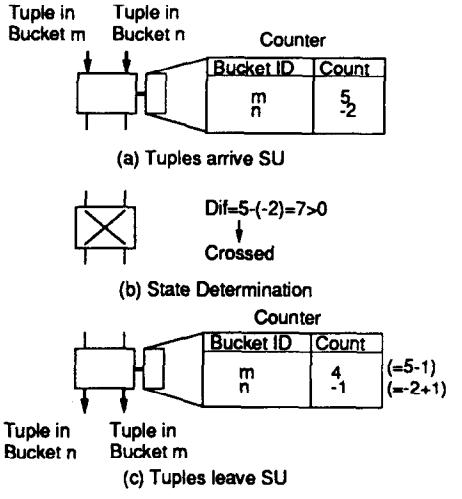

Figure 11: State Determination in Switching Unit

6.2 Performance Evaluation of the Bucket Flattening Network

We examine the effectiveness of the bucket flattening algorithm described above. Flatness of bucket distribution is measured as follows: first the standard deviation of bucket distribution is calculated for each bucket, and the mean is taken over all buckets. This value is referred to as mean standard deviation (MSD). We compared MSDs before and after flattening using simulations. Bucket ID is randomly determined following uniform distribution in the simulations.

Figure 12, in which the horizontal axis indicates the number of tuples and the vertical axis denotes MSD, shows a simulation result when $N = 8$, $B = 128$, and T varies from 512 to 16K tuples. N, B and T represent the number of PMs, the number of buckets and the number of tuples per PM. The high initial MSD shows the need for flattening. The MSD after flattening by the network is very low indicating that bucket distribution is almost flat across the PMs. The MSD after flattening is almost constant regardless of the number of tuples. Figure 13 shows the variation due to the number of PMs. Simulation for the case, in which $B = 128, T = 8k$ and N varies from 2 to 64, indicates that the network flattens bucket distribution very well.

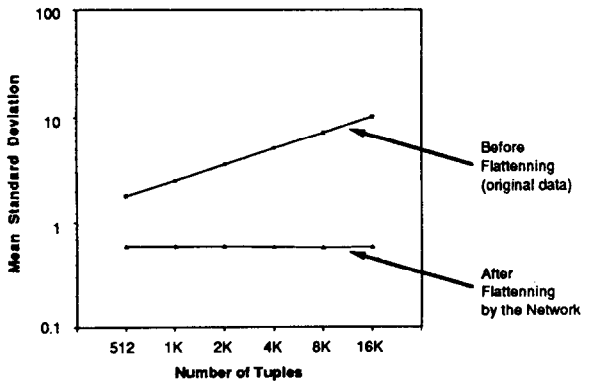

Figure 12: Effect of Flattening (N=8, B=128)

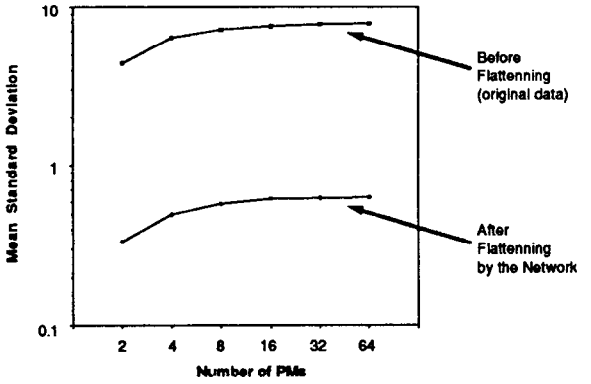

Figure 13: Effect of Flattening (B=128, T=8k tuples)

6.3 Implementation Issues

Here we briefly examine two issues on network implementation. First, the logic needed to determine SU's state is very simple, the additional hardware for SU is a comparator and counters for the same number of buckets. The control mechanism is implemented in a LSI chip. Secondly, not to become a bottle neck of the processing, we designed the SU chip so that the bandwidth of the network matches the data transfer rate of the disk. Such bandwidth can be easily achieved using the current technology; for instance, the bandwidth of each line is 6 MB/s in the Teradata DBC/1012 where the network embeds the sort function [Nec88,Tan88].

7 Performance Evaluation of the Bucket Spreading Strategy in SDC

In this section, we present the performance evaluation of the bucket spreading strategy using simulations.

7.1 Simulation Model

Simulations followed the model described hereafter. The two source relations are the same size, i.e. $|R| = |S| = 128$K tuples, and they are horizontally partitioned so that the number of tuples in each PM is equal to that in the others. In addition, both of the relations are assumed to follow the same distribution, which means $|R_i| = |S_i|$ (R_i and S_i represent the ith bucket of relation R and S). The size of each bucket is determined by a Zipf-like distribution [Tur87]. The size of each bucket is given by:

$$|R_i| = \frac{|R|}{i^z \cdot \sum_{b=1}^{B} \frac{1}{b^z}}$$

When $z = 1$ this equals a Zipf distribution, and when $z = 0$ it is a uniform distribution. Distribution of income is known to follow a Zipf-like distribution of $z = 0.5$. Bucket sizes are illustrated in Figure 14 when B(the number of buckets) $= 256$.

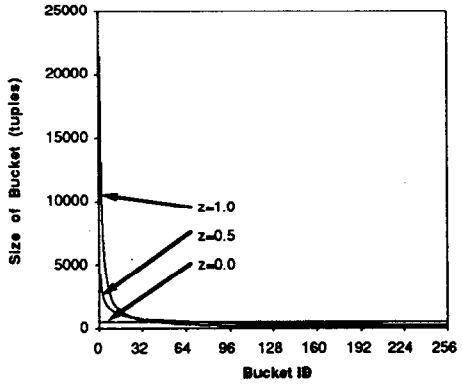

Figure 14: Bucket Size Distribution (Zipf-like Distribution)

The capacity of each PM's memory is 4K tuples. Bucket size tuning is carried out as described in [Nak88,Kit89c]. The bucket flattening omega network, as explained in Section 6, is used for the in-

terconnection network. Although the network functions as an ordinary omega network when the bucket converging strategy is adopted, the bucket spreading strategy utilizes the bucket flattening mechanism. The transfer rates for both modes are assumed to be equal. When data conflict in the network occurs in the bucket converging strategy, one tuple is randomly selected to be transferred, and the others are discarded and retransmitted. It should be remembered that there is no conflict in the bucket spreading strategy described in Section 6.1.

Performance is measured by the number of tuple I/Os. Tuple I/Os for storing the result relation are not counted because they are the same for both strategies. In the bucket converging strategy, because overall processing time is determined by the PM which has the largest number of tuples, the I/O is counted for all PMs and the maximum is used as the resultant count. In these simulations, because selectivity and joinability are assumed to be 1.0, all tuples are read twice and written once in a GRACE join.

7.2 Effects of Data Skew

We first measure the effect of data skew where the number of PMs is set to 8. The size of the source relations is 128K tuples, so every PM stores 16k tuples from each relation. Thus, the minimum of I/O count is 96K($= 2 * 3 * 16$K) tuples. The number of buckets is fixed at 256.

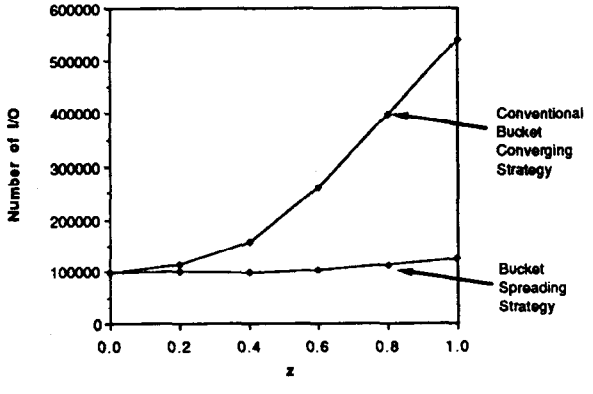

Figure 15: Effect of Data Skew (1)

Figure 15 and Figure 16 show the simulation results, where the horizontal axis indicates the value of z of the Zipf-like distribution and the vertical axis denotes the number of I/Os. In Figure 15, we use the bucket converging strategy of the worst case scenario, in that buckets are allocated as follows: the ith bucket is allocated to the the $\lceil (i \times N)/B \rceil$th PM. When the

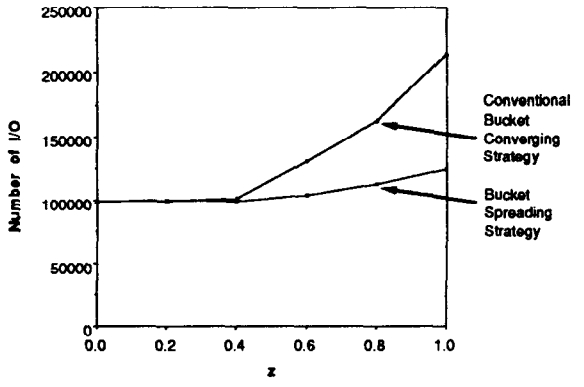

Figure 16: Effect of Data Skew (2)

two relations are distributed uniformly ($z = 0.0$), both strategies show nearly the same performance, that is, almost equals to the minimum number of I/Os. When the distribution becomes unbalanced, the number of I/Os steeply increases in bucket converging, because most large buckets concentrate on one PM, becoming a bottleneck. Conversely, the bucket spreading strategy attains almost the same performance for every distribution. In Figure 16, the degree of skew in bucket converging is decreased as follows; the ith bucket is allocated to the the $(((i-1) \bmod N) + 1)$th PM. Although the performance of bucket converging is improved, as a distribution becomes skewed, the number of I/Os is still increased and much more than that for bucket spreading. These results confirm that the bucket spreading strategy is quite effective for data skew.

7.3 Effects of the Number of PMs

Next, we examine the effect of increasing the number of PMs. In simulations, we change the number of PMs from 1 to 64 [5]. When the number of PMs increase, the number of buckets should be increased. However, the size of the portion of the relation stored in each PM is decreased conversely, which means the number of buckets can be decreased. So the number of buckets is set to the same number, 256, for all simulations.

Figure 17 and Figure 18, where the horizontal axis is the number of PMs and the vertical is the speed up rate, show simulation results for $z = 1.0$. In these figures, the bucket spreading strategy realizes an almost linear increase of the speed up to the number of PMs

[5] In the current version of SDC, because each PM has 4 processors, the actual number of processors varies from 4 to 256.

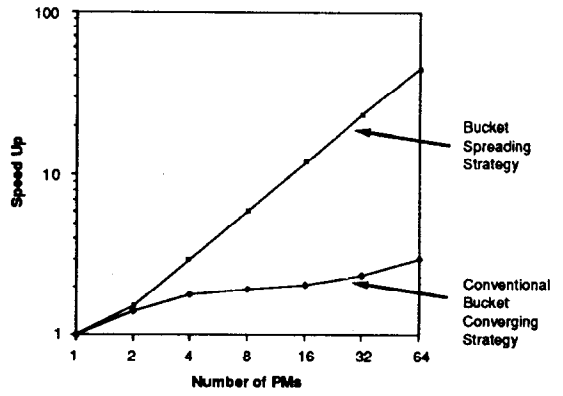

Figure 17: Comparison of Speed Up (1)

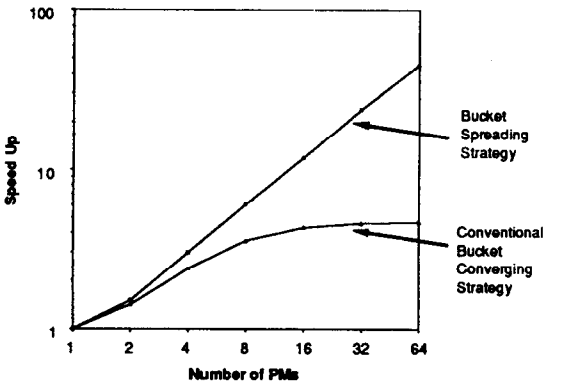

Figure 18: Comparison of Speed Up (2)

for both cases, confirming that the bucket spreading strategy is quite robust for data skew. On the other hand, the speed up is not linear in the bucket converging strategy (Figure 17 is for the worst case, and Figure 18 is for the improved one). The speed up is saturated at the level where the number of PMs is 8 or 16, because, when data is highly skewed, one huge bucket virtually determines the overall performance. In Zipf distribution, the largest bucket holds 21401 tuples (about 1/6 of whole relation), and other buckets allocated to the same PM become negligible when the number of PMs is large. Therefore the speed up is saturated in bucket converging.

8 Conclusion

In this paper, we proposed a parallel hash join method which is robust for data skew. In the conventional method, based on the bucket converging strategy in which a bucket is statically allocated to a PM according to its bucket ID, the performance is severely affected by data distribution. To solve this problem, we have proposed the bucket spreading strategy. Each bucket is partitioned evenly into subbuckets which are placed across all the PMs, and then these subbuckets are collected to the appropriate PM. Since buckets are dynamically allocated to PMs after bucket size tuning, we expect efficient processing against data skew. In SDC, because the bucket flattening omega network carries out bucket partitioning and flattening, PMs are not burdened with additional overhead necessary for flat bucket distribution.

We evaluated the performance of the bucket spreading strategy using simulations. When data distribution is uniform, there is no significant difference between the bucket converging strategy and the bucket spreading strategy. However, when data distribution becomes unbalanced, the bucket spreading strategy showed almost the same performance as with a balanced distribution, while the performance of the bucket converging strategy was severely degraded. Furthermore, almost linear speed up was observed for the bucket spreading strategy. Simulation results confirmed the effectiveness of the bucket spreading strategy.

Application of the bucket spreading strategy to other hash join methods such as Hybrid hash [DeW84] and Dynamic Hybrid GRACE hash [Nak88,Kit89c] is being simulated, and we will present the results in an upcoming paper. SDC now is operational as a single PM system. Performance measurement based on real data will be presented in the near future.

References

[Bra84] K. Bratbergsengen. Hashing Methods and Techniques for Main Memory Database Systems. In *Proc. of the 10th Int. Conf. on VLDB*, pp. 323–333, 1984.

[Bra89] K. Bratbergsengen and T. Gjelsvik. The Development of the CROSS8 and HC16 – 186 Parallel Computers. In *Proc. of the 6th Int. Conf. on Database Machines*, pp. 359–372, 1989.

[DeW84] D. J. DeWitt, *et. al.* Implementation Techniques for Main Memory Database Systems. In *ACM SIGMOD '84*, pp. 1–8, 1984.

[DeW85] D. J. DeWitt and R. Gerber. Multiprocessor Hash-based Join Algorithms. In *Proc. of the 11th Int. Conf. on VLDB*, pp. 151–164, 1985.

[DeW86] D. J. DeWitt, *et. al.* GAMMA, A High Performance Dataflow Database Machine. In *Proc.*

of the 12th Int. Conf. on VLDB, pp. 228–237, 1986.

[Ger86] R. H. Gerber. *Dataflow Query Processing Using Multiprocessor Hash-partitioned Algorithms.* Technical Report 672, University of Wisconsin, 1986.

[Hir90] S. Hirano, M. Kitsuregawa, *et. al.* Architecture of SDC, the Super Database Computer. In *Proc. of the JSPP '90*, pp. 137–144, 1990 (in Japanese).

[Kit83] M. Kitsuregawa, H. Tanaka, and T. Motooka. Application of Hash to Database Machine and its Architecture. *New Generation Computing*, 1(1):66–74, 1983.

[Kit84] M. Kitsuregawa, *et. al.* Architecture and Performance of Parallel Relational Database Machine GRACE. In *Proc. of the 14th Int. Conf. on Parallel Processing*, pp. 241–250, 1984.

[Kit87] M. Kitsuregawa, M. Nakano, L. Harada, and M. Takagi. Functional Disk System for Relational Database. In *Proc. of the 3th Int. Conf. on Data Engineering*, pp. 88–95, 1987.

[Kit87b] M. Kitsuregawa, *et. al.* Design and Implementation of High Speed Pipeline Merge Sorter with Run Length Tuning Mechanism. In *Proc. of the 5th Int. Workshop on Database Machines*, pp. 144–157, 1987.

[Kit89] M. Kitsuregawa, M. Nakano, and M. Takagi. Query Execution for Large Relations on Functional Disk System. In *Proc. of the 5th Int. Conf. on Data Engineering*, pp. 159–167, 1989.

[Kit89b] M. Kitsuregawa and W. Yang. Implementation of LSI Sort Chip for Bimodal Sort Memory. In *Proc. of the Int. Conf. on VLSI*, pp. 285–294, 1989.

[Kit89c] M. Kitsuregawa, *et. al.* The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method. In *Proc. of the 15th Int. Conf. on VLDB*, pp. 257–266, 1989.

[Lak88] M. S. Lakshmi and P. S. Yu. Effect of Skew on Join Performance in Parallel Architecture. In *Proc. of Int. Symp. on Databases in Parallel and Distributed Systems*, pp. 107–120, 1988.

[Lak89] M. S. Lakshmi and P. S. Yu. Limiting Factors of Join Performance on Parallel Processors. In *Proc. of the 5th Int. Conf. on Data Engineering*, pp. 488–496, 1989.

[Law75] D. H. Lawrie. Access and Alignment of Data in an Array Processor. *IEEE Transaction on Computers*, C-24(12):1145–1155, 1975.

[Mit89] Mitsubishi Electric Corp. RDB Processor GREO. 1989.

[Nak88] M. Nakayama, M. Kitsuregawa, and M. Takagi. Hash-partitioned Join Method Using Dynamic Destaging Strategy. In *Proc. of the 14th Int. Conf. on VLDB*, pp. 468–478, 1988.

[Nec88] P. M. Neches. The Ynet: An Interconnect Structure for a Highly Concurrent Data Base Computer System, In *Proc. of 2nd Symp. on the Frontiers of Massively Parallel Computation*, pp. 429–435, 1988.

[Omi89] E. R. Omiecinski and E. T. Lin. Hash-based and Index-based Join Algorithms for Cube and Ring Connected Multicomputers. *IEEE Transaction on Knowledge and Data Engineering*, KDE-1(3):329–343, 1989.

[Rie78] D. Ries and R. Epstein. *Access Path Selection in a Relational Database Management System.* Technical Report UCB/ERL-M78/22, UC Berkeley., 1978.

[Sch89] D. A. Schneider and D. J. DeWitt. A performance Evaluation of Four Parallel Join Algorithms in a Shared-nothing Multiprocessor Environment. In *ACM SIGMOD '89*, pp. 110–122, 1989.

[Sha86] L. D. Shapiro. Join Processing in Database Systems with Large Main Memories. *ACM Transactions on Database Systems*, 11(3):239–264, 1986.

[Sto86] M Stonebraker. A Case for Shared-nothing. *IEEE Database Engineering*, 9(1), 1986.

[Ter88] Teradata Corp. *DBC/1012 Data Base Computer Concepts and Facilities.* Technical Report C02-0001-05, Teradata Corp., 1988.

[Tan88] The Tandem Performance Group. A Benchmark of NonStop SQL on the Debit Credit Transaction. In *ACM SIGMOD '88*, pp. 337–341, 1988.

[Tur87] C. Turbyfill. *Comparative Benchmark of Relational Database systems.* PhD thesis, Cornell University, September 1985.

[Val84] P. Valduriez and G. Gardarin. Join and Semijoin Algorithms for a Multiprocessor Database Machines. *ACM Transactions on Database Systems*, 9(1):133–161, 1984.