Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning

Kien A. Hua

Department of Computer Science
University of Central Florida
Orlando, FL 32816-0362
U. S. A.

Chiang Lee

Institute of Information Engineering
National Cheng Kung University
Tainan, Taiwan, 70101
Republic of China

Abstract

Shared nothing multiprocessor architecture is known to be more scalable to support very large databases. Compared to other join strategies, a hash-based join algorithm is particularly efficient and easily parallelized for this computation model. However, this hardware structure is very sensitive to the data skew problem. Unless the parallel hash join algorithm includes some load balancing mechanism, skew effect can deteriorate the system performance severely.

In this paper, we propose two *skew avoidance techniques* and one *skew resolution method*. In particular, three new parallel hash join algorithms are presented. We developed an analytical model to study the effectiveness of these algorithms. The performance study indicates that the proposed techniques offer substantial improvement over the conventional strategies in the presence of data skew. It is also interesting to observe that the skew avoidance techniques provide join strategies that are robust against data skew; where as the skew resolution method offers an adaptive join strategy that outperforms the conventional algorithms for any skew condition.

1 Introduction

There are several architectures for designing multiprocessor database computers. However, The multicomputer model [1], is most popularly used for its scalability to support very large databases [2, 3, 4, 5, 6]. The hardware structure of this computation model consists of a number of *processing nodes* (PN) interconnected through a communication network. Each PN has its own private memory and dedicated disk drives. This architecture is also known as message passing or shared nothing [7] multiprocessor structure. In this environment, the relations are typically declustered into fragments and spreaded across the PNs. Since in this model each PN processes the portion of the database on its disks, the degree of parallelism is determined by the distribution of data in the system. When serious data skew occurs, balancing the load on all the PNs is needed to ensure high system performance.

The join operation has been the most intensively studied among the relational operations for this structure. Since join operations are generally very expensive, a query optimizer typically defers the join operations until the data reduction process is performed by less costly operations, such as selections and projections. For this reason, the distribution of the input relations to a join operation is typically hard to predict. Most likely the data fragments result from the earlier data reduction operations will vary in size, and a load balancing mechanism will be necessary to ensure the efficient performance of the succeeding join operation [8].

Several parallel join algorithms have been proposed. Among them, hash-based algorithms [9, 10] are particularly suitable for the multicomputer model. In these strategies the relations are hashed (partitioned) into buckets, and each bucket is allocated to a distinct PN. Since tuples of a relation in one bucket are never joined with tuples of the other relation in other buckets, matching buckets can be joined independently in parallel by all PNs. The effectiveness of these schemes depends on the uniformity of the tuple distribution. Although they have been demonstrated to be very effective, these techniques do not guarantee the balanced workload among the local join operations. When severe fluctuation occurs among the bucket sizes, the skew effect deteriorates the performance of the join operations. A *Bucket Spreading Parallel Hash Join* (BSJ) was introduced in [11] to correct this problem.

Unlike the conventional hash-based join algorithms that statically assigned the buckets to the PN prior to the data partitioning process, BSJ algorithm defers the bucket allocation until the data partitioning process is completed. Since the distribution of the data among the buckets then becomes known, the buckets can be assigned to the PNs dynamically based on the bucket sizes to ensure a balanced data load at each PN. However, when buckets are to be allocated dynami-

cally, the destination of a tuple cannot be determined during partitioning because "which PN to handle the tuple" is not assigned yet. BSJ solves this problem by spreading the original buckets across all PNs, and collecting them later to the appropriate destinations just prior to the join operations. To avoid a bottleneck during the bucket collection phase, BSJ uses a special Omega network [12] to ensure that the buckets evenly spread across all PNs. The implementation of this switch includes at each switch element, as many number of counters as there are number of buckets involved. Such a switch design may face two problems. First, it does not scale well with the rest of the system. As the number of PNs increases to support larger databases, the number of buckets increases causing the increase in the complexity of the switch elements (i.e., need more counters). Even when the system does not yet have a large number of PNs, the switch element must have enough number of counters to anticipate system growth. In other words, the complexity in the switch design could penalize the smaller system configurations. Secondly unless other operations can be designed to take advantage of the data spreading feature of the functional switch, the complexity in the switch design may affect the communication performance of the switch, and consequently the performance of the whole system. Depending on the transaction mix, it may not be worthwhile to improve the performance of joins on the expense of other operations. Finally the complexity explodes when we consider a multiuser environment. Managing different sets of counters for different users becomes quite expensive.

In [13], we proposed an adaptive load balancing strategy using *Partition Tuning*. In this approach, a relation is organized as a set of data cells (data fragments), and balanced work load is achieved by reassigning data cells from overflow PNs to underflow PNs using a *Best Fit Decreasing* strategy. In this paper, we discuss the load balancing technique in the context of join operation. We will introduce three parallel join algorithms with dynamic load balancing capability using the partition tuning concept:

1. **Tuple Interleaving Parallel Hash Join (TIJ)**: This scheme is similar to BSJ algorithm. However, instead of relying on a specially designed switch to maintain uniform bucket spreading across PNs, it uses software control to interleave the tuples among PNs as they are being spread to their destinations.
2. **Adaptive Load Balancing Parallel Hash Join (ABJ)**: This technique introduces an additional step to the conventional parallel hash join algorithms [9, 10]. This added process relocates the excess buckets from the larger PNs to the smaller PNs attempting to balance the data load prior to the join operations.
3. **Extended Adaptive Load Balancing Parallel Hash Join (ABJ$^+$)**: In this strategy, each PN hashes its local portion of the relations into local subbuckets. These local subbuckets are stored back in the local disks. After this partitioning process is complete, the distribution of the data in each bucket can be computed. The matching local subbuckets are then collected to their final destination to form the corresponding buckets. The gathering of the subbuckets are based on the bucket sizes to ensure balanced work load for each of the PNs during the join phase.

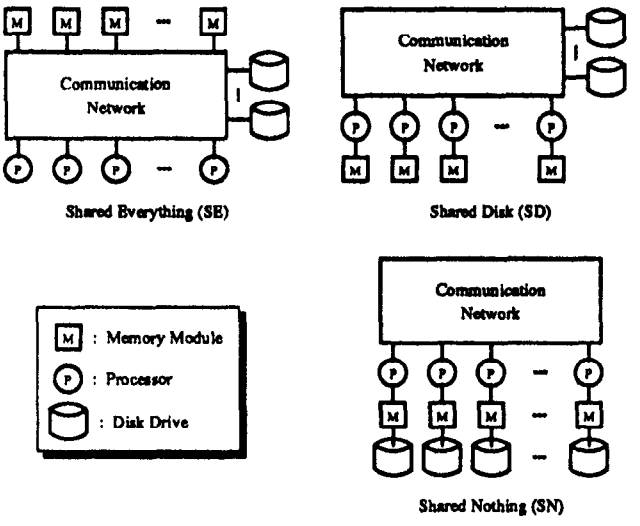

Figure 1: Architectures for multiprocessor database computers.

In the following sections we will describe these algorithms in more details. A performance model will be introduced and it will be used to compare the performance of these strategies.

The rest of this paper is organized as follows. The design environment and assumptions are described in Section 2. Some problems in hash-based join strategy is discussed, and the proposed join algorithms are presented in Section 3. In Section 4, we introduce an analytical model and discuss the performance comparison of the proposed schemes. Finally, we offer our conclusions in Section 5.

2 System Environment

Basically, there are three different architectures for multiprocessor database computers: *Shared Everything* (SE), *Shared Disk* (SD), and *Shared Nothing* (SN) (Figure 1). There has been a lot of debate in the database management community about which architecture is most suitable for a database management system. The coherency control problem limits the number of processors that can efficiently cooperate in a SE or a SD system. In a SE system, processors share a single global memory address space. The shared memory is typically a physically distributed memory to accommodate the aggregate demand on the shared

memory from a large number of processors. An interconnection network is usually used to allow any processor to access any memory module. This communication network, however, increases the memory-access latency. The performance of conventional processors is quite sensitive to memory-access latency. If the memory-access latency exceeds one instruction time, the processor must idle until the storage cycle completes. A popular solution to this problem is to have cache memory with each processor. However, the use of caches requires a mechanism to ensure cache coherency. As we increase the number of processors, the number of messages due to cache coherency control (i.e., cross interrogation) increases. Unless this problem can be solved, scaling a SE system into the range of hundreds of processors will be impractical. Similarly in a SD system, it is obvious that inter-processor coherency control is necessitated due to the caching of the shared database pages in main memory database buffers. The buffer invalidation problem tends to be the limiting factor for the size of a SD system.

In contrast, data coherency control is not a problem in a SN system. Besides, the processor and memory are physically localized in a node, and memory-access latency is not a problem. Nevertheless, SN systems are very sensitive to the data skew problem [8]. When the data are seriously skewed, rebalancing the data load among the PNs is necessary to resume good system performance.

From the above discussion, it seems that there is no absolute winner. When the individual processors are very powerful as in the case of mainframe computers, since we don't need a lot of processors in order to achieve the required performance level a small number of very high performance processors can be interconnected in a shared memory structure to avoid the communication overhead and the data skew problems (e.g., IBM 3090 Model 600). On the other hand if the applications demand a performance level far exceeding the capability of typical mainframe systems, then a microprocessor-based SN structure offers a solution for handling very large databases (e.g., iPSC/2 version of GAMMA [3], NonStop SQL [4], [5], Super Database Computer (SDC) [11], DBC/1024 [6]). In this paper we assume the latter environment and discuss dynamic load balancing strategies for parallel join operations in these systems. We would like to note that a combination of the three generic parallel architectures can also be employed to benefit from the advantages of each scheme. One such hybrid structure was discussed independently in [14] and [11].

3 New Parallel Hash Join Algorithms

There are two major problems associated with hash-based parallel join algorithms:

1. **Bucket overflow**: In hash-based join algorithms, the size of each bucket should be smaller than the memory capacity. However, nonuniform distribution of the join attribute values occasionally generates *bucket overflow*, in which the sizes of the buckets exceed the memory capacity. The performance diminishes because it requires extra I/O to repartition the buckets into smaller fragments so that each will fit in the memory.
2. **Data skew**: The performance of the conventional parallel hash join algorithms relies on the randomizing hash function to redistribute the tuples of the join relations evenly across all PNs in the system. Their performance degrades when the join attribute values of the relations are non-uniformly distributed [9, 8, 15]. This phenomenon is known as the *data skew* problem, in which some PNs have significantly more tuples than the remaining PNs.

To overcome the bucket overflow problem, Hybrid Hash Join [10] uses a second hash function, h_2, to stream the overflow tuples to a temporary file on disk. In other words, h_2 redistributes the overflow bucket between an in-memory hash table and overflow buckets on disk. GRACE Hash Join [9] tries to avoid the bucket overflow problem by splitting the relations into a large number of smaller buckets, and then these small buckets are combined into buckets to fit the memory capacity. This process is referred to as *bucket tuning* in [9]. Although these conventional parallel hash join algorithms effectively resolves the bucket overflow problem, no mechanism is provided to avoid the data skew effect. This problem is addressed in the *Bucket Spreading Parallel Hash Join* (BSJ) algorithm [11] by deferring the bucket allocation process until after the data partitioning procedure is completed. The delay allows the buckets to be allocated to PNs based on the bucket sizes to ensure a balanced data load in the system.

BSJ algorithm provides an effective parallel algorithm for performing join under skew conditions. This scheme, however, requires an expensive specially designed network to support the bucket spreading mechanism. As discussed in Section 1, the additional complexity in the communication network may also cause the following problems:

1. It does not scale well with the rest of the system.
2. It could degrade the communication performance of the network.

In this section we present three new efficient algorithms that require only conventional hardware. Therefore, they are immediately applicable to many existing parallel database computers.

3.1 Tuple Interleaving Parallel Hash Join

The purpose of the special hardware used in [11] is to ensure the uniform distribution of each bucket across all PNs. Alternatively, the bucket spreading effect can be achieved by software control. One way to achieve

the uniform bucket distribution effect is to send each tuple to the PN currently containing the smallest subbucket among all the subbuckets of its bucket. Thus to determine the destination of a tuple, each PN has to keeps track of data distribution on every PN. As criticized in [11], this approach would require a high volume of data transfer between PNs, resulting in considerable performance degradation. For this reason, a highly functional Omega network is used in [11] to resolve the problem. Alternatively, this problem can be avoided by having each PN interleaving the tuples among the PNs as they were spread out from a bucket. This tuple interleaving strategy is described in more details in the *Tuple Interleaving Parallel Hash Join* (TIJ) algorithm given below.

1. **Split Phase**: R and then S are partitioned in parallel. Each PN independently divides its partition of each of the relations R and S stored therein into p buckets where p is considerably larger than the number of PNs in the system. During this partitioning process, tuples belonging to each of these buckets are spread across the PNs. The spreading is done by interleaving the consecutive tuples from each bucket among the PNs in the system. For each PN, the ith tuple of a bucket is sent to the $(((i-1) \bmod N)+1)$th PN, where N is the number of PNs in the system. Since this spreading strategy guarantees that each bucket is spread evenly among PNs, the N subbuckets of each bucket such derived, therefore, should be uniform in size.
2. **Bucket Tuning Phase**: Since buckets are distributed evenly among the PNs, the distribution of subbuckets in each PN is representative of the distribution of buckets among all PNs. Therefore a predetermined coordinating PN can decide how to tune the size of buckets to fit the memory capacity based only on its local distribution of subbuckets. The remaining PNs can then tune their subbuckets accordingly as directed by the coordinator.
3. **Partition Tuning Phase**: The coordinating PN groups the buckets into N equal partitions, and allocates each partition to a distinct PN. The bucket-to-PN mapping information is then broadcast to all the remaining PNs. Each PN then forms its partition by sending and receiving tuples as directed in the mapping information. Even after bucket size tuning, the buckets vary slightly in size. To reduce the effect of this nonuniformity, we can sort the buckets according to size. Then the sorted buckets are allocated to the PNs in a round-robin fashion [11]. In the case that the buckets vary greatly in size, a partition tuning strategy as described in [13] can be employed, in which the tuning algorithm is performed by first sorting in descending order the bucket according to size. The buckets are then allocated to the PNs in the sorted order. The assignment is done by allocating the currently largest buckets in the sorted list to the currently smallest PN. The bucket is then removed from the list. This process is repeated until the sorted list is exhausted. Hereafter, we will refer to this process as the *Best Fit Decreasing* strategy.
4. **Join Phase**: Each PN performs the local joins of respectively matching buckets.

We note that TIJ algorithm is a variation of the BSJ algorithm originally proposed in [11]. Except for the Split Phase, the remaining three phases of TIJ are similar to the corresponding steps in the BSJ algorithm. Our bucket spreading strategy employs software control at each PN instead of using special hardware as proposed in the original BSJ algorithm. The overhead associated with the software control is essentially negligible. For instance, a simple round-robin strategy is sufficient to keep track of the next destination for each bucket in a PN. This mechanism can also be implemented in hardware by providing a simple *spreading processor* at each of the input to the communication network. This spreading processor maintains a set of round-robin counters, one for each bucket involved. When a tuple is received by the spreading processor, the content of the round-robin counter is used as the destination for the tuple transfer. The counter is then incremented by one (modulo N, where N is the number of PNs). Comparing this scheme to the hardware design proposed in [11], we see two advantages. First, the number of counters used in the proposed scheme is proportional to N; whereas the complexity of the scheme proposed in [11] is $O(N \log N)$. Second, decoupling the complexity of the bucket spreading hardware from the communication media is a good idea to maintain the high communication throughput for all operations. An alternative to this design is to emulate the round-robin mechanism in the communication processor at each PN. Most today's communication processor should be capable of performing this additional simple task without becoming a bottleneck in the system.

We note also that the data transfer as described in the TIJ algorithm does not have to be performed at the tuple level. In practice, each PN maintains N outgoing buffers. Tuples belonging to the same destination are piggybacked to the same buffer, and the buffer is sent to its destination when it is full. In addition, if an appropriate network (e.g., Omega) is used the transfer of buffers can be synchronized to follow the cyclic shift pattern [16] in order to avoid access conflict. The four possible cyclic shift patterns for a 4-PN system is depicted in Figure 2.

Furthermore, if the system is designed for a multiuser environment it is advantageous to employ a partitionable communication network [17] (e.g., multistage shuffle-exchange networks) so that the system can be reconfigured dynamically into smaller "independent" parallel engines to serve different queries if necessary. In such an environment, not all base relations are large; smaller relations need to store on fewer

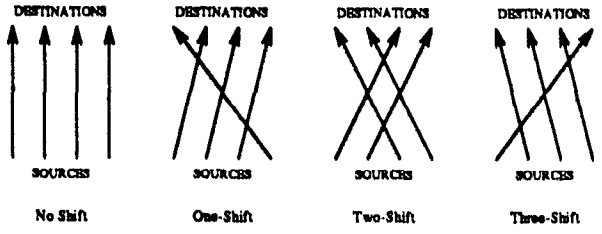

Figure 2: Cyclic shift patterns.

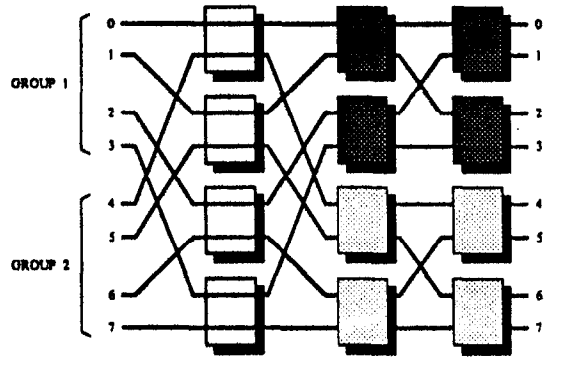

Figure 3: Cube network of size eight partitioned into two subnetworks of sizes four.

PNs only to reduce the skew effect and communication overhead. They are assigned to the PNs in such a way to take advantages of the partitioning properties of the communication network. Similarly for non-base relations, the intermediate operations can be centralized to as little or to as great an extent as is appropriate to maximize the system throughput. These individual operators can be scheduled to run on different partitions of the hardware in order to eliminate network contention among concurrent queries. In addition, the data transmission due to an individual query is synchronized within a hardware partition to follow a cyclic shift pattern [16] in order to avoid regional access conflict. For example, Figure 3 shows a cube network of size eight partitioned into two independent subnetworks. The first group consists of PNs connected to ports 0 to 3. The second group consists of PNs connected to ports 4 to 7. Since the original network supports the cyclic shift permutations without conflicts, the partitioning properties of the cube network guarantee that each of the two subnetworks will have all the connection properties of a cube network including the cyclic shift communication patterns. In a multiuser environment, we are concerned about system throughput in addition to response times. The use of a partitionable network reduces network contention among concurrent queries, and consequently improves the throughput of the multiprocessor system.

3.2 Adaptive Load Balancing Parallel Hash Join

A major advantage of the TIJ algorithm is that the work load is very balanced at each PN for all phases of the algorithm. A drawback of this strategy, however, lies in the fact that it shuffles the whole tuple space entirely during their load balancing process regardless of the degree of data skew. When the skew condition is mild, this strategy tends to result in unnecessary communication and computation overhead. The *Adaptive Load Balancing Parallel Hash Join* (ABJ) algorithm given in the following addresses this problem.

1. **Split Phase**: R and then S are partitioned into a large number of buckets in parallel. Each bucket is statically allocated to a PN as in GRACE Hash Join algorithm. Each tuple in a bucket is collected to the corresponding PN through the interconnection network.
2. **Partition Tuning Phase**: This phase consists of two stages.
   (a) **Bucket Retaining Stage**: For each relation, each PN_i retains n buckets using some best-fit strategy so that their aggregated size satisfies the following condition:

$$\sum_{j=1}^{n} |B_{ij}| \leq \frac{|A|}{N} \quad and \quad \sum_{j=1}^{n+1} |B_{ij}| > \frac{|A|}{N}$$

   where $|A|$ is the size of the corresponding relation, and $|B_{ij}|$ is the size of the jth bucket of PN_i. The remaining buckets not retained in this stage are termed the *excess buckets*.
   (b) **Bucket Relocating Stage**: Each PN reports its size and the size of the excess buckets to a designated coordinating PN. The coordinator then use these information to reallocate the excess buckets to the underflow PNs using the Best Fit Decreasing strategy described in the last subsection. Once the destination of the excess buckets has been determined, this information is broadcast to the PNs, and the excess buckets are physically collected by the PNs accordingly.
3. **Bucket Tuning Phase**: Each PN combines the small buckets to form more optimal size join buckets.
4. **Join Phase**: Each PN performs the local joins of respectively matching buckets.

We note in the Bucket Retaining Stage that each PN should try to retain the larger buckets since the Best Fit Decreasing Strategy employed by the following Bucket Relocation Stage works better with smaller buckets. Comparing the Bucket Relocating Stage of this algorithm with the Partition Tuning Phase of the TIJ algorithm, we see that the former attempts to keep the data transfer to minimal, whereas the latter strategy shuffles on average $\frac{N-1}{N}$ of the tuples of

each relation. When the skew condition is mild, the number of tuples at each PN varies very slightly and only little data movement will be needed during the Bucket Relocating Phase of the ABJ algorithm. In this sense, ABJ is capable of adapting dynamically to the degree of skew. For this reason, we call this strategy Adaptive Load Balancing Parallel Join algorithm. Since the skew conditions are typically mild (that is why the conventional hash join algorithms are so effective e.g., GRACE Hash Join, Hybrid Hash Join), we expect the proposed ABJ algorithm to have good performance for many applications.

3.3 Extended Adaptive Load Balancing Parallel Hash Join

When the skew condition is serious, disk overflow may occur during the Split Phase of the ABJ strategy. In addition, the severe skew condition causes the transfer of tuples concentrate on the skewed PN resulting in communication hot spot. Moreover, it further degrades the performance since these excess tuples must be reallocated to the underflow PNs during the Partition Tuning Phase. The *Extended Adaptive Load Balancing Parallel Hash Join* (ABJ$^+$) algorithm presented in this subsection corrects this problem by deferring the tuple transfer until the Partition Tuning Phase. The details of the algorithm is outlined bellow.

1. **Split Phase**: Each PN partitions its portion of each relation into considerably small subbuckets. Each subbucket is stored back in the local disks (no disk overflow).
2. **Partition tuning**: Each PN reports the sizes of its subbuckets to a designated coordinating PN. The coordinator adds up the sizes of the matching subbuckets distributed across the PNs to derive the sizes of the corresponding buckets. The coordinator then allocates the buckets to the PNs using the following strategy:
   (a) The buckets are sorted into descending order according to their sizes.
   (b) The buckets are then allocated to the PNs in the sorted order. For each bucket, it is assigned to the PN with the largest matching subbucket among the qualified PNs[1] in order to minimize communication overhead. The size of that PN is then updated to reflect the addition of the new bucket. When the size of a PN satisfies the following condition:

$$\sum_{j=1}^{n} |B_{ij}| \leq \frac{|A|}{N} \quad and \quad \sum_{j=1}^{n+1} |B_{ij}| > \frac{|A|}{N}$$

   where $|A|$ is the size of the corresponding relation, and $|B_{ij}|$ is the size of the jth bucket of PN_i, it is disqualified from consideration for any further bucket allocation.

[1] PNs which have a matching subbucket

   This iterative process continues until all buckets are allocated. Once this process is complete, the allocation information is broadcast to all PNs, and the subbuckets are physically collected accordingly to their respective destination to form the corresponding local buckets.
3. **Bucket Tuning Phase**: Each PN combines the small buckets to form more optimal size (i.e., fit the memory capacity) join buckets.
4. **Join Phase**: Each PN performs the local joins of respectively matching buckets.

We note in this strategy that no tuple travels across a PN boundary more than once during the entire algorithm. Furthermore, it takes advantage of the already balanced portion of the tuple space to minimize the disk I/O and communication overheads. In this sense, it is capable of adapting dynamically to the current condition of the tuple distribution in the system.

4 Performance Analysis

In this section, we develop an analytical model for the four parallel join algorithms: TIJ, ABJ, ABJ$^+$and GRACE. First we describe the performance model, and present the cost function for each of the join algorithms. These cost functions are then used to perform sensitivity analysis with respect to three parameters: degree of data skew, I/O bandwidth, and communication bandwidth. In this paper, GRACE algorithm which does not handle skew problems is used as the reference for the comparison of the three proposed techniques.

4.1 Performance Model

The following parameters are designed for cost evaluation. They are similar to those used in [8].

- **Workload Parameters:**
  $|R|$: Relation size in tuples for each of the joining relations.
  $|P_s|$: Size in tuples of the skewed partition.
  $|P_u|$: Size in tuples of each of the remaining unskewed partitions.
  σ: The degree of data skew which is defined as $\sigma = \frac{|P_s|-|P_u|}{|P_s|}$.
  t: Size in bytes of each tuple.
- **System Parameters:**
  N: Number of PNs in the system.
  M: Memory capacity in bytes for each PN.
  μ: CPU processing rate in million-instructions-per-second (MIPS).
  ω_{io}: I/O bandwidth between a processor and its secondary storage.
  ω_{comm}: Effective communication channel bandwidth per PN.

I_{cpu}: CPU pathlength for processing a tuple in any step of the join operation.

- **Measurement Parameters**

T_{split}: Time cost in seconds due to a Split Phase.

T_{parti}: Time cost in seconds due to a Partition Tuning Phase.

T_{bucket}: Time cost in seconds due to a Bucket Tuning Phase.

T_{join}: Time cost in seconds due to a Join Phase.

T_{split_io}: Time cost in seconds for disk accesses during a Split Phase.

T_{split_cpu}: Time cost in seconds for processing tuples during a Split Phase.

T_{split_comm}: Time cost in seconds for transferring data among PNs during a Split Phase.

T_{parti_io}: Time cost in seconds for disk accesses during a Partition Tuning Phase.

T_{parti_comm}: Time cost in seconds for transferring data among PNs during a Partition Tuning Phase.

T_{join_hash}: Time cost is seconds for building a hash table for a bucket in memory.

T_{join_probe}: Time cost in seconds for probing the in-memory hash table using tuples from the matching bucket.

T_{TIJ}: Time cost in seconds for joining two relations using the TIJ algorithm.

T_{ABJ}: Time cost in seconds for joining two relations using the ABJ algorithm.

T_{ABJ+}: Time cost in seconds for joining two relations using the ABJ$^+$algorithm.

T_{GRACE}: Time cost in seconds for joining two relations using the GRACE Hash Join algorithm.

In this study, the system environment as described in Section 2 is assumed. we assume that initially both relations are horizontally partitioned and evenly distributed among the PNs (i.e., the sizes of all partitions of a relation are equal). In the join operation, we further assume that under the skew condition the relations are evenly distributed among all PNs, except one (i.e., The skewed PN) which has excess data. In other words, for each of the joining relations the skewed PN has $\sigma \times 100\%$ more tuples than each of the remaining PNs. σ is called the degree of skew, and it has value ranging between 0 and 1. This assumption is based on the fact that the most seriously skewed PN constitutes the bottleneck in our shared nothing environment. It dictates the performance of the parallel execution regardless of those less severely skewed PNs. Therefore, it is sufficient to assume a single skewed PN for our purpose. Based on this assumption, we thus have:

$$|P_s| + (N-1)|P_u| = |R| \Longrightarrow \begin{cases} |P_s| = \frac{|R|}{1+(N-1)(1-\sigma)} \\ |P_u| = \frac{(1-\sigma)|R|}{1+(N-1)(1-\sigma)} \end{cases}$$

$|P_s|$and $|P_u|$derived here will be used in the following subsections to compute the time costs for the parallel join algorithms.

4.2 Cost Functions

In this subsection, we present the cost functions for the join algorithms based on the described architecture and workload. Since partial overlap can exist between the phases of the join algorithms, the total join cost, T_{total} is bounded by:

$$\max(T_{phase\_1}, T_{phase\_2}, \cdots T_{phase\_n}) \leq T_{total}$$
$$\leq T_{phase\_1} + T_{phase\_2} + \cdots + T_{phase\_n}$$

Where max represents the maximum function. Similarly, each phase consists of several steps (disk accesses, tuple processing, communication). Overlapping of those steps is also achievable. A performance upper bound and lower bounds for the phases can be derived accordingly. In our study, we made the following assumptions:

- The overlap within each phase is perfect. The system is assumed to include a separate I/O processor and a separate communication processor which allow the overlap among disk I/O, CPU computation, and data communication [5].
- The overlap between two phases is not allowed. That is, a simple barrier type synchronization [18] is used between the phases to guarantee the correct parallel execution.

Therefore, the join time can be computed as:

$$T_{total} = T_{phase\_1} + T_{phase\_2} + \cdots + T_{phase\_n}$$

4.2.1 Time Cost for GRACE Hash Join

GRACE Parallel Hash Join consists of two distinct phases. Its cost is computed below:

$$T_{GRACE} = T_{split} + T_{join}$$

$$T_{split} = \max(T_{split\_io}, T_{split\_cpu}, T_{split\_comm})$$

$$T_{split\_io} = \frac{2|R|}{N} \cdot \frac{t}{\omega_{io}} + \frac{2|P_s|t}{\omega_{io}}$$

$$T_{split\_cpu} = \frac{2|R|}{N} \cdot \frac{I_{cpu}}{\mu}$$

$$T_{split\_comm} = \left(2|P_s| + \frac{N-2}{N} \cdot \frac{2|R|}{N}\right) \cdot \frac{t}{\omega_{comm}}$$

$$T_{join} = \left\lceil \frac{|P_s| \cdot t}{M} \right\rceil \cdot (T_{join\_hash} + T_{join\_probe})$$

$$T_{join\_hash} = \max\left(\frac{M}{\omega_{io}}, \frac{M}{t} \cdot \frac{I_{cpu}}{\mu}\right)$$

$$T_{join\_probe} = \max\left(\frac{M}{\omega_{io}}, \frac{M}{t} \cdot \frac{I_{cpu}}{\mu}\right)$$

$$\Longrightarrow \quad T_{join} = 2 \cdot \left\lceil \frac{|P_s|t}{M} \right\rceil \cdot \left[\max\left(\frac{M}{\omega_{io}}, \frac{M}{t} \cdot \frac{I_{cpu}}{\mu}\right)\right]$$

4.2.2 Time Cost for Tuple Interleaving Parallel Hash Join Algorithms

TIJ algorithm consists of four phases. Its time cost is computed below:

$$T_{TIJ} = T_{split} + T_{parti} + T_{bucket} + T_{join}$$

$$T_{split} = \max(T_{split\_io}, T_{split\_cpu}, T_{split\_comm})$$

$$T_{split\_io} = 2 \cdot \left(\frac{2|R|t}{N\omega_{io}}\right) \qquad T_{split\_cpu} = \frac{2|R|}{N} \cdot \frac{I_{cpu}}{\mu}$$

$$T_{split\_comm} = 2 \cdot \frac{N-1}{N} \cdot \frac{2|R|t}{N\omega_{comm}} = \frac{4(N-1)|R|t}{N^2 \cdot \omega_{comm}}$$

$$T_{bucket} \approx 0$$

$$T_{parti} = \max(T_{parti\_io}, T_{parti\_comm})$$

$$T_{parti\_io} = 2 \cdot \frac{N-1}{N} \cdot \frac{2|R|t}{N\omega_{io}} = \frac{4(N-1)|R|t}{N^2 \cdot \omega_{io}}$$

$$T_{parti\_comm} = 2 \cdot \frac{N-1}{N} \cdot \frac{2|R|t}{N\omega_{comm}} = \frac{4(N-1)|R|t}{N^2 \cdot \omega_{comm}}$$

$$T_{join} = \left\lceil \frac{|R|t}{N \cdot M} \right\rceil \cdot (T_{join\_hash} + T_{join\_probe})$$

$$T_{join\_hash} = \max\left(\frac{M}{\omega_{io}}, \frac{M}{t} \cdot \frac{I_{cpu}}{\mu}\right)$$

$$T_{join\_probe} = \max\left(\frac{M}{\omega_{io}}, \frac{M}{t} \cdot \frac{I_{cpu}}{\mu}\right)$$

$$\Longrightarrow \quad T_{join} = 2 \cdot \left\lceil \frac{|R|t}{N \cdot M} \right\rceil \cdot \max\left(\frac{M}{\omega_{io}}, \frac{M}{t} \cdot \frac{I_{cpu}}{\mu}\right)$$

4.2.3 Time Cost for Adaptive Load Balancing Parallel Hash Join Algorithm

ABJ algorithm consists of four phases. Its time cost is computed below:

$$T_{ABJ} = T_{split} + T_{parti} + T_{bucket} + T_{join}$$

$$T_{split} = \max(T_{split\_io}, T_{split\_cpu}, T_{split\_comm})$$

$$T_{split\_io} = \frac{2|R|t}{N\omega_{io}} + \frac{2|P_s|t}{\omega_{io}} \qquad T_{split\_cpu} = \frac{2|R|}{N} \cdot \frac{I_{cpu}}{\mu}$$

$$T_{split\_comm} = \left\{\frac{N-1}{N} \cdot \frac{2|R|}{N} + \left[2|P_s| - \frac{1}{N} \cdot \frac{2|R|}{N}\right]\right\} \cdot \frac{t}{\omega_{comm}}$$

$$T_{parti} = \max(T_{parti\_io}, T_{parti\_comm})$$

$$T_{parti\_io} = \frac{N-1}{N} \cdot (|P_s| - |P_u|) \cdot \frac{t}{\omega_{io}}$$

$$T_{parti\_comm} = \frac{N-1}{N} \cdot (|P_s| - |P_u|) \cdot \frac{t}{\omega_{comm}}$$

As in TIJ algorithm, we let $T_{bucket} \approx 0$. Also, the time cost for the Join Phase, T_{join}, is the same as that of TIJ algorithm (Section 5.2.2).

4.2.4 Time Cost for Extended Adaptive Load Balancing Parallel Hash Join Algorithm

ABJ$^+$ algorithm consists of four phases. Its time cost is computed below:

$$T_{ABJ+} = T_{split} + T_{parti} + T_{bucket} + T_{join}$$

$$T_{split} = \max(T_{split\_io}, T_{split\_cpu}, T_{split\_comm})$$

$$T_{split\_io} = 2 \cdot \left(\frac{2|R|t}{N\omega_{io}}\right) \qquad T_{split\_cpu} = \frac{2|R|}{N} \cdot \frac{I_{cpu}}{\mu}$$

$$T_{split\_comm} = 0$$

$$T_{parti} = \max(T_{parti\_io}, T_{parti\_comm})$$

$$T_{parti\_io} = 2 \cdot \frac{N-1}{N} \cdot \frac{2|R|t}{N\omega_{io}} = \frac{4(N-1)|R|t}{N^2 \cdot \omega_{io}}$$

$$T_{parti\_comm} = 2 \cdot \frac{N-1}{N} \cdot \frac{2|R|t}{N\omega_{comm}} = \frac{4(N-1)|R|t}{N^2 \cdot \omega_{comm}}$$

Finally $T_{bucket} \approx 0$ and the time cost for the Join Phase can be computed as in TIJ algorithm (Section 5.2.2).

4.3 Sensitivity Analysis

With the model we developed, we are able to do the performance sensitivity study for the proposed parallel join algorithms under different parameters. The values of the parameters we used in our study are listed in the following:

1. **Workload Parameters:**
   - Relation size ($|R|$): 1 million tuples each
   - Tuple size (t): 200 Bytes per tuple
   - Degree of skew (σ): varies from 0.1 to 1.0
2. **System Parameters:**
   - Number of PNs (N): 64
   - Memory Capacity (M): 2 MBytes per PN
   - CPU processing rate (μ): 20 MIPS
   - I/O bandwidth (ω_{io}): varies from .25 to 4 MBytes/Second per PN
   - Effective communication channel Bandwidth (ω_{comm}): varies from .25 to 4 MBytes/Second per PN
   - Instruction pathlength (I_{cpu}): 1,000 instructions

Among these parameters, we select the degree of data skew, disk I/O bandwidth, and communication bandwidth for the sensitivity analysis.

We note that the I/O bandwidth is set to 4 MBytes/sec which is typical for the industry standard SCSI bus. The communication bandwidth for each port of the communication network is also set to 4 MBytes/sec to match the data transfer rate of the disk controller. We could have set the bandwidth of the communication network to a fixed number independent of the number of PNs in the system. In practice,

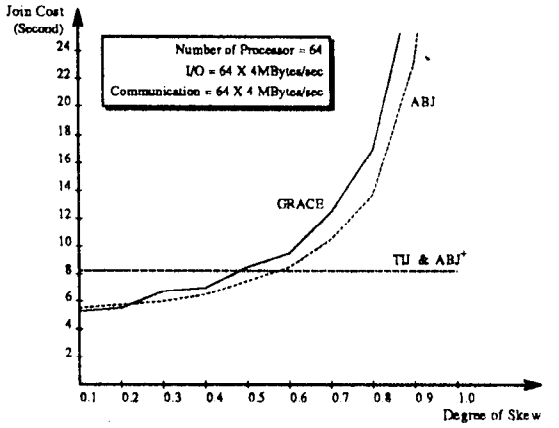

Figure 4: Skew effect on the parallel join algorithms.

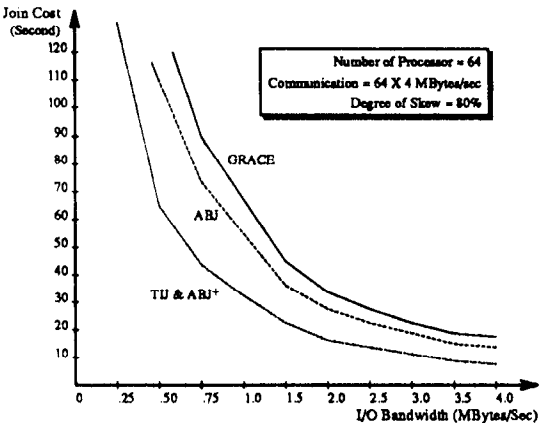

Figure 5: Effect of I/O bandwidth on the parallel join algorithms.

this is the case for systems using a common bus for interprocessor communication. For instance, Tandem NonStop Computer consists of 2 to 16 PNs connected by two high-speed 20Mb/sec buses [19]. Our model represents communication network topologies whose bandwidth increases with the increase in the number of communication ports. Crossbar switches, multi-stage networks are examples of this category. For simplicity, we assume the effective bandwidth of the communication network increases linearly with the number of PNs.

To prevent the processor from becoming a bottleneck, the processing rate of each PN is set to 20 MIPS which is derived as follows:

$$\mu = I_{cpu} \cdot \frac{\omega_{io}}{t} = 20MIPS$$

We present the results of the sensitivity analysis in the following subsections.

4.3.1 Skew Effect

The three parallel hash join algorithms proposed in this papers can be loosely grouped into two categories: TIJ and ABJ+are *skew avoidance techniques*, whereas ABJ is a *skew resolution method*. This classification is based on the fact that TIJ and ABJ+maintain balanced data load for each PN at all time to prevent the skew problems. On the other hand, ABJ attempts to resolve the skew problems as they arise.

The effect of the data skew on the parallel join algorithms is plotted in Figure 4. The corresponding data are also given wherein. It is interesting to observe that the algorithms TIJ and ABJ+are robust against the skew effect. Their time costs are constant regradless of the degree of skew. Furthermore the two algorithms share the same performance curve in this study. This is due to the fact that I/O operation can be overlapped with the data communication process in our model. Although ABJ+reduces one round of data transfer in the Split Phase as compared to TIJ, this communication reduction does not have an effect on the savings unless the communication network becomes the bottle neck in the system ($\omega_{comm} < \omega_{io}$). In order to perform the sensitivity analysis with respect to the skew effect, we designed the perfect parameters (i.e., $\omega_{comm} = \omega_{io}$) to avoid this bottleneck.

The drawbacks of TIJ and ABJ+are the I/O and communication overheads associated with the deferred bucket allocation strategies. In TIJ, subbuckets of a bucket are temporarily saved at all PNs, and must be collected to their appropriate PN later. In ABJ+, the subbuckets of a bucket are stored back to the local disks during the Split Phase, and must be reloaded in order to send them to their final destination. Due to these disk I/O and communication overheads, TIJ and ABJ+cannot provide savings over GRACE algorithm until $\sigma > 45\%$ (i.e., one of the 64 PNs has 2.8% of the total tuples) as depicted in Figure 4. Similarly, ABJ performs better than TIJ and ABJ+for $\sigma < 60\%$. Since the skew conditions are typically mild for many applications, it makes ABJ attractive for some environments. Nevertheless, both GRACE and ABJ are sensitive to data skew, the robustness of the TIJ and ABJ+against skew conditions makes them uniquely appealing to highly parallel database systems.

In comparing ABJ to GRACE, we see that ABJ performs better for $\sigma > 20\%$ (i.e., one of the 64 PNs has 1.94% of the total tuples). In practice, at the beginning of the Partition Tuning Phase of ABJ, we can decide whether to proceed with the tuning process based on the skew condition. With this extension, it is possible to implement ABJ to perform at least as well as, or better than GRACE for any skew condition.

4.3.2 Effect of I/O Bandwidth

In the sensitivity study with respect to the degree of data skew, we assume that the hardware design is "perfect" - the processors, the I/O subsystems, and the communication processors are tuned for the join operation. In this and the following subsection, we are

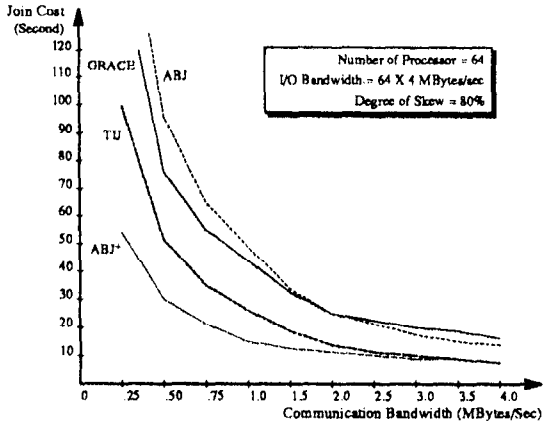

Figure 6: Effect of communication bandwidth on the parallel join algorithms.

interested in an environment that is less than ideal. In this subsection, we study how the performance of the I/O subsystems can affect the algorithms. In particular, how the overhead due to load balancing is related to the I/O bandwidth. In order to compare the three load balancing strategies over a wide range of I/O bandwidths, we purposely set the skew condition to a large number, namely $\sigma = 80\%$. This means that one of the 64 PNs has 7.3% of the total tuples according to our model. The results of this study is ploted in Figure 5. The corresponding data are also given wherein.

This study shows that TIJ and ABJ$^+$are the better choices when the I/O bandwidth is low. Furthermore, the savings due to the load balancing strategies with respect to GRACE algorithm increases as the I/O bandwidth decreases. This suggests that load balancing will become even more critical when the performance gap between the processor technology and I/O technology widen - a phenomenon we are observing today.

Finally, we also observe here that TIJ and ABJ$^+$share the same performance curve. This is due to the fact that $\omega_{io} < \omega_{comm}$ in this sensitivity study.

4.3.3 Effect of Communication Bandwidth

We are also interested in the effect of the communication bandwidth on the proposed join algorithms. Again, we set $\sigma = 80\%$ for this sensitivity analysis. The results of this study is ploted and the corresponding data are given in Figure 6.

We observe that ABJ$^+$is the best scheme in this study. We also note that although TIJ and ABJ$^+$perform equally well in Figure 4 and Figure 5, ABJ$^+$outperforms TIJ in systems with limited communication capability as shown in Figure 6. From this study, it suggests that ABJ$^+$is the better approach of the two proposed skew avoidance techniques.

Finally, comparing Figure 5 to Figure 6 we see that load balancing is even more critical in a system with limited communication capability than in a system with inadequate I/O performance. In Figure 4, we made the assumption that the effective bandwidth of the communication network increases linearly with the number of PNs. This is over optimistic. In practice, the rate of increase in communication bandwidth decreases as the number of PNs increases, and we expect to see the benefits of the proposed schemes at even lower skew conditions than those suggested in Figure 4.

5 Conclusion

In this paper, we discussed dynamic load balancing strategies for parallel hash join algorithms. In particular, we proposed three new parallel hash join strategies. They are categorized as follows:

1. **Skew Resolution Technique:**
   - *Adaptive Load Balancing Parallel Hash Join Algorithm* (ABJ)
2. **Skew Avoidance Techniques:**
   - *Tuple Interleaving Parallel Hash Join Algorithm* (TIJ)
   - *Extended Adaptive Load Balancing Parallel Hash Join Algorithm* (ABJ$^+$)

We developed a performance model, and did sensitivity analysis to study the effect of data skew, I/O bandwidth, and communication bandwidth on the proposed schemes. The results of our study indicate that:

- ABJ algorithm should be used if the degree of skew is mild.
- ABJ$^+$algorithm should be used if the degree of skew is significant.
- TIJ algorithm should be used in lieu of ABJ$^+$algorithm if
  1. the relations are very seriously skewed initially, and
  2. the communication bandwidth is sufficiently large.

Overall, the proposed schemes are able to provide savings over conventional parallel hash-based join algorithms even at low skew conditions.

In addition to the comparison results, we also observed that load balancing is more critical to systems with limited I/O and communication capabilities. With the cost of communication hardware remains expensive, and the performance gap between microprocessor technology and I/O technology widen rapidly, the benefits of the proposed join strategies are evident for today's multicomputer database management systems.

Finally, although we assumed a barrier synchronyzation between join phases in our performance model for

simplicity, in practice techniques used in the popular Hybrid Hash Join algorithm [20] can be employed to overlap the join phases whenever possible. This strategy is particularly beneficial when the memory capacity is large. In addition, the concept of *cell* as described in [13] can also be used, in which tuples are organized into cells to facilitate the environment for efficient execution of the Best Fit Decreasing algorithm during partition tuning.

References

[1] Danial A. Reed and Richard M. Fujimoto. *Multicomputer Networks: Message-Based Parallel Processing*. Scientific Computation. MIT Press, 1987.

[2] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):4–24, 1990.

[3] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, 1990.

[4] Susanne Englert, Jim Gray, Terry Kocher, and Praful Shah. A benchmark of nonstop sql release 2 demonstrating near-linear speedup and scaleup on large databases. Technical Report 89.4, Tandem Computer Inc., 1989.

[5] Kien A. Hua and Honesty C.Young. Designing a highly parallel database server using off-the-shelf components. In *Proceedings of The International Computer Symposium*, Hsinchu, Taiwan, December 1990.

[6] Teradata Corporation, Los Angesles, Calafornia. *Teradata DBC/1012 Data Base Computer Concepts and Facilities*, release 3.1 edition, 1988. Teradata Document C02-0001-05.

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

[8] S. Lakshmi and P. S. Yu. Effect of skew on join performance in parallel architectures. In *Proceedings of International Symposium on Databases in Parallel and Distributed Systems*, pages 107–117, Austin, Texas, December 1988.

[9] M. Kitsuregawa, H. Tanaka, and T. Moto-oka. Application of hash to database machine and its architecture. *New Generation Computing*, 1(1):66–74, 1983.

[10] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of the SIGMOD Conference*, pages 110–121, 1989.

[11] M. Kitsuregawa and Yasushi Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (sdc). In *Proceedings of 16th International Conference on Very Large Data Bases*, pages 210–221, Brisbane, Australia, August 1990.

[12] D. H. Lawrie. Access and alignment of data in an array processor. *IEEE Transaction on Computers*, c-24(12):1145–1155, December 1975.

[13] Kien A. Hua and Chiang Lee. An adaptive data place scheme for parallel database computer systems. In *Proceedings of 16th International Conference on Very Large Data Bases*, pages 493–506, Brisbane, Australia, August 1990.

[14] Kien A. Hua, Chiang Lee, and Jih-Kwon Peir. A high-performance hybrid architecture for concurrent query execution. In *Proceedings of Symposium on Parallel and Distributed Processing*, Dallas, Texas, December 1990.

[15] S. Lakshmi and P. S. Yu. Limiting factors of join performance on parallel processors. In *Proceedings of 5th International Conference on Data Engineering*, pages 488–496, 1989.

[16] Jacques Lenfant. Parallel permutations of data: A benes network control algorithm for frequently used permutations. *IEEE Transaction on Computers*, c-27(7):637–647, July 1978.

[17] Howard Jay Siegel. The theory underlying the partitioning of permutation networks. *IEEE Transaction on Computers*, c-29(9):791–800, September 1980.

[18] Harry F. Jordan. A special purpose architecture for finite element analysis. In *Proceedings of 1978 International Conference on Parallel Processing*, pages 263–266, 1978.

[19] Robert Holbrook. Nonstop sql - a distributed relational dbms for oltp. In *Proceedings of Compson '88*, San Francisco, California, Februaray 1988.

[20] D. J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 SIGMOD Conference*, pages 1–8, Boston, MA, June 1984.