Performance Analysis of a Load Balancing Hash-Join Algorithm for a Shared Memory Multiprocessor

Edward Omiecinski

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332
U. S. A.

Abstract

Within the last several years, there has been a growing interest in applying general multiprocessor systems to relational database query processing. Efficient parallel algorithms have been designed for the join operation but usually have a failing in that their performance deteriorates greatly when the data is nonuniform. In this paper, we propose a new version of the hash-based join algorithm that balances the load between the processors, for any given bucket, in a shared everything environment. We develop an analytical model of the cost of the algorithm and implement the algorithm on a shared memory multiprocessor machine. We also perform a number of experiments comparing our model with our empirical results.

1. Introduction

Applying multiprocessor machines to database query processing has been an active area of research. The motivation for using a multiprocessor machine stems from the fact that databases are getting larger and that an adequate level of performance, e.g. response time, is required. Most of the proposed multiprocessor database systems are based on a shared nothing [BF87,DGGHKM86,KO90] architecture. That is, where each processor has its own local memory and processors communicate via message passing. In addition, a disk or set of disks is connected to each processor, allowing processors to read/write to a disk in parallel. There have also been proposed systems that follow a shared everything [DKT90,MR89,OS90,QI88,SKPO88] paradigm. In these schemes memory and disks are shared by all processors. Once again, a disk (or set of disks) would have its own I/O controller so that a read/write to different disks can be done in parallel.

For relational database systems, there have been a number of algorithms developed for implementing the join operation in parallel [BF87,DG85,QI88]. They include the nested-loop method [BF87,SD89], the sort-merge method [QI88,SD89] and the hash-join method [QI88,SD89]. The performance of the algorithms is usually predicted by analytical modelling or simulation. In the performance analysis it is assumed that the data (relations) are uniformly distributed. However, as pointed out in [LY88,SD89], a non-uniform data distribution (also referred to as data skew) can have a severe affect on the performance of the join algorithm.

In this paper we develop a hash-join algorithm that is robust in the face of data skew. It is specifically designed for a shared everything architecture. We present some of the previous work that has been done to devise a join algorithm that will be immune to data skew in the next section. In section 3, we present our hash-join algorithm that balances the load when the data is skewed. In section 4, we present an analytical cost model for our algorithm and in section 5, we show the results of implementing our algorithm on a shared memory multiprocessor machine.

2. Previous Work

In [WDY90], the shared nothing model of parallelism is assumed and a parallel join algorithm based on the sort-merge method is presented to handle data skew. The algorithm is based on the divide-and-conquer approach. It adds an extra scheduling phase to the usual sort, transfer and join phases. During the scheduling phase, an optimization algorithm is used, which takes the output of the sort phase and determines how the join is to be divided into multiple tasks and how those tasks are to be assigned to proces-

sors so as to balance the load. They present an analytical model of their algorithm's performance and show that it achieves a good load balancing for the join processing phase in a CPU-bound environment.

In [KO90], a robust hash-join based algorithm is devised for a specific parallel database computer architecture. This architecture is also based on a shared nothing model. Instead of their previous approach [KNHT90] of allocating buckets to processors, they dynamically allocate buckets to processors so as to balance the load. They propose a bucket spreading strategy which partitions buckets into fragments and in a subsequent phase these fragments are assigned to processors. The bucket spreading strategy is similar to the idea of disk striping. They also use a specific network structure, i.e., an omega network, to assist in the bucket spreading strategy. They present a simulation model of their system and show that the performance of their algorithm is not affected very much by the presence of data skew. The cost of writing the result of the join to a file and to disk is not considered.

In [SD89], the effect of limited data skew on four different join algorithms is examined. They conclude that the performance of the hash-based join algorithms degrade when the join values of the inner relation are highly skewed and that a non-hash-based algorithm should be used in those cases, e.g. sort-merge. However, the double skew case was not considered.

Some work has appeared in the recent literature dealing with hash-join algorithms for a shared everything architecture [LTS90,QI88]. In [QI88], they present a large set of parallel algorithms, including hash-join, for implementing a join operation on a shared memory database machine. An analytical model of the various algorithm's performance is presented. However, the problem of data skew was not considered.

In [LTS90], they examine only hash-based join algorithms for a general purpose shared memory multiprocessor. The amount of available memory is assumed to be proportional to the number of processors. In this approach a global hash table is built for each bucket. They use a locking mechanism to provide exclusive access for a write to this hash table but they allow multiple reads to occur simultaneously. They provide an analytical model of the total processing time for their join algorithms. As in the previously mentioned work in this section, data skew is not considered.

3. Hash Join Algorithms for a Shared Everything Architecture

The basic architecture that we assume is that of a shared everything machine. Each processor shares common memory with other processors and each processor may have some local (nonshared) memory as well. Processors also share secondary storage devices. We assume that for secondary storage, we have the same number of disk drives as we do processors. We do not have to limit our approach to this but we had to make a decision regarding this feature and this seemed reasonable. The processors, disks and memory are linked by an interconnection network. We assume that disk reads/writes of different pages can be done concurrently as can memory accesses to distinct variables. Although main memory sizes are increasing rapidly, we assume that neither of the two relations to be joined will fit entirely in main memory.

The hash-join algorithm that we consider for a shared memory multiprocessor is the GRACE hash-join method [KTM83]. This algorithm will also form the basis for our load balancing hash-join algorithm. Since the number of processors will be relatively small in a shared memory multiprocessor, the number of buckets produced will be greater than the number of processors.. Hence, a single processor will handle multiple buckets. The GRACE hash-join algorithm can be summarized by the following phases for performing R ⋈ S.

I. Read relation R, partition the tuples into buckets based on a given hash function applied to the joining attribute value and write those buckets to secondary storage;

II. Read relation S, partition the tuples into buckets based on the same hash function as used in prior phase and write those buckets to secondary storage;

III. For each bucket created in phase I, read the R bucket from secondary storage, build an in memory hash table using some hash function applied to the joining attribute value, read the corresponding S bucket from secondary storage and for each tuple probe the hash table for an R tuple with the matching value for the

joining attribute. The join result is formed by the matching tuples.

One can see that if the data for each relation is partitioned across all the disks, then each phase can be done in parallel. As seen in [LTS90], one has to be careful in phases I and II that multiple processors do not try to write to the same bucket at the same time. In [LTS90], they used locks to avoid the problem. However, when the data is highly skewed the lock conflict rate can be quite high for both phases. For our adaptation of this algorithm, we avoid the use of locks. In phase I (or phase II) each processor determines the bucket that a tuple belongs in but does not write it to the bucket at that time. Instead that information is stored in a small array. The array is small since we are processing a page of a relation at a time on each processor. Hence, the number of entries in the array is equal to the number of tuples in a page. Each processor sorts its array in increasing order of bucket number and builds a small index on that information, i.e., linking a bucket number with the tuples that belong to that bucket. After that, the processors are assigned buckets. The tuples for each given bucket will be placed in a buffer page by only one processor. To accomplish this, a given processor will access a distinct entry in the index of each of the processors. For example, if a processor is responsible for bucket 0, then that processor will access entry 0 in the index created by each of the processors. With this scheme only one buffer page per bucket is needed and there is no contention between processors.

In contrast to the approach in [LTS90], one hash table is not built at a time and all processors do not probe that hash table in parallel during phase III. As previously mentioned, if the data is highly skewed, then building and probing a single hash table in parallel can result in a high conflict rate that degrades performance. Instead, if there are n processors then hash tables for n buckets will be created in parallel and probed in parallel.

Our load balancing hash-join approach is similar to the approach in [KO90] but with the following notable differences: our method is designed for a shared everything system as opposed to a shared nothing system, we make use of no special hardware and we assign a bucket to one or more processors based on a first-fit decreasing heuristic, similar to that used for the bin-packing problem [HS78]. In addition, we develop an analytical model of our algorithm's cost and implement the algorithm so that we can validate our model.

For our load balancing hash-join approach, we need to modify the three phases and to add a scheduling phase prior to phase III. In phase I, we keep a count of the number of tuples that hash into each bucket for relation R. In addition, since one bucket may need to be processed by several processors in phase III, we must stripe the pages of a bucket across the disks. This will allow us to read in a portion of one bucket in parallel and to process each portion in parallel. We need to do the same for phase II but for relation S. Having the size (in number of tuples) for each bucket we can decide on the number of processors to handle each bucket and the schedule of buckets to processors. For a given bucket, we use the maximum of the sizes of the corresponding R and S buckets for the following calculation. To determine the number of processors that are needed to handle a bucket, we calculate the size of a uniform bucket, *usize*, i.e., as if the tuples were uniformly distributed to buckets. We then divide each actual bucket size by *usize* and take the ceiling of that result, which yields the number of needed processors per bucket, *processors_per_bucket*. The scheduling phase, which precedes phase III, sorts the *processors_per_bucket* information in increasing order. Next a schedule is produced by using a first-fit decreasing heuristic to allocate buckets to individual steps in the schedule so that the number of steps is minimized. In the first-fit decreasing heuristic, buckets are allocated to a step in non-increasing order of the number of processors needed. If the required number of processors for the bucket to be scheduled is not available in the current set of steps in the schedule, then a new step is added to the schedule and the bucket is allocated to the processors in that new step.

Once the scheduling phase is complete, which is done by a single processor, phase III may begin using the schedule as a driver. The schedule is a global array and as such is accessible by all processors. Using the schedule information, each processor involved with a given bucket can read a disjoint subset of the pages of that bucket since the pages are striped across the disks and can build an in-memory hash table of the tuples on those pages. Now, since there does not exist just a single hash table for a given bucket but, let say n hash tables where n is the number of processors needed to process the

bucket, the tuples in the corresponding S bucket must be processed against each of those n hash tables. Once again, since the pages of the S bucket have been striped across the disks, a disjoint subset of those pages can be processed by each of the n processors. At this point the process resembles the nested-loop join method since a page read (by processor i) and processed against hash table i has to be processed against the remaining $n-1$ hash tables. This is accomplished in a circular fashion, i.e., processor i processes a data page with hash tables $i,i+1,...,n,0,...,i-1$ in that order. The circular approach is also used for reading pages from the disks so that contention is minimized due to multuple processors trying to read distinct pages from the same disk at the same time.

Table I. Model Parameters

Parameter	Description/ Value
|R|	size of relation R = 1000 (in pages)
|S|	size of relation S = 1000 (in pages)
{R}	number of R tuples = 10000
{S}	number of S tuples = 10000
JV_R	number of unique join values in relation R = 1000
t_R	number of tuples per page of relation R = 10
t_S	number of tuples per page of relation S = 10
P	number of processors = 2 thru 9
B	number of buckets = 25
D	number of disks = 2 thru 9
IO	time to perform an I/O operation = 24 ms
Comp	time to compare two attributes = 0.007 ms
Assign	time to assign a value to a variable = 0.007 ms
C	sorting constant = 2
F	scheduling constant = 2
Hash	time to compute hash value of an attribute = 0.015 ms
Move	time to move a tuple in memory = 0.040 ms
Rskew	fraction of R tuples with skew data value = 0.1 to 0.4
Sskew	fraction of S tuples with skew data value = 0.1 to 0.4

4. Analytical Models for Hash Join Algorithms

In this section we present the analytical models for both hashing schemes. To make the modelling more tractable we make a few assumptions. We should note that these assumptions are not restrictions on the algorithms. The main assumption regards the data skew. We assume that only one bucket contains a disproportionate number of tuples, which is a percentage of the total number of tuples in the relation. The other tuples are uniformly distributed across the remaining buckets. To accomplish this we have a single data skew value for the joining attribute and we generate a specified fraction of the tuples with that value. The remaining tuples will actually be distributed across all of the buckets, including the skew bucket. In addition, when we consider the double skew case, the corresponding buckets in the other relation have the same structure. This is in agreement with the assumptions made in other work [LY88]. We also assume that there is enough main memory to process any given bucket, since we are primarily interested in how data skew affects load balancing and not how data skew may cause overflow in a hash bucket. Table I describes the different parameters and their values used with the analytical models. They are similar to the values found in [LTS90,QI88,Sha86].

We will first present the model for the basic hash-join approach and then for the load balancing approach. Since the processors are working in parallel, the one processor that takes the longest time to finish in each step will dictate the overall cost of the algorithm. That processor is the one that is assigned the skew bucket. As already mentioned, that processor will also handle an equal share of the other buckets. Hence, the cost formulas will be derived for just that processor. In **phase I**, we have the following steps:

(a) Read pages of R relation:

$$\left\lceil \frac{|R|}{P} \right\rceil \cdot IO$$

(b) Find bucket address of tuple:

$$\left\lceil \frac{|R|}{P} \right\rceil \cdot t_R \cdot Hash$$

(c) Store address of tuple and bucket address in an array:

$$2 \cdot \left\lceil \frac{|R|}{P} \right\rceil \cdot t_R \cdot Assign$$

(d) Sort array by bucket number:

$$C \cdot \left\lceil \frac{|R|}{P} \right\rceil \cdot t_R \log_2 t_R \cdot (Compare + Assign)$$

(e) Build index on bucket number:

$$\left\lceil \frac{|R|}{P} \right\rceil \cdot (t_R \cdot Compare + 2 \cdot B \cdot Assign)$$

(f) Move tuples to their associated buckets: Note, based on our assumptions, one processor will handle the skew bucket as well as $\left\lceil \frac{B}{P} \right\rceil - 1$ other buckets since the bucket to processor assignment is static and since all processors will handle approximately the same number of buckets. So the skew value contributes $\lceil Rskew \cdot \{R\} \rceil$ tuples to the skew bucket. Since the other tuples are uniformly distributed across all B buckets, each bucket will have $\left\lceil (1 - Rskew) \cdot \frac{\{R\}}{B} \right\rceil$ tuples. So the total number of tuples in any nonskew bucket can be defined as

$$t_{nonskewbucket} = \left\lceil (1 - Rskew) \cdot \frac{\{R\}}{B} \right\rceil$$

and the total number of tuples in the skew bucket can be defined as

$$t_{skewbucket} = \lceil Rskew \cdot \{R\} \rceil + t_{nonskewbucket}$$

So we have the following cost for this step:

$$2 \cdot \left\lceil \frac{B}{P} \right\rceil \cdot P + t_{skewbucket} \cdot Move$$
$$+ \left(\left\lceil \frac{B}{P} \right\rceil - 1 \right) \cdot t_{nonskewbucket} \cdot Move$$

(g) Write pages of buckets to disk:

$$\left[\left\lceil \frac{t_{skewbucket}}{t_R} \right\rceil + \left(\left\lceil \frac{B}{P} \right\rceil - 1 \right) \cdot \left\lceil \frac{t_{nonskewbucket}}{t_R} \right\rceil \right] \cdot IO$$

Phase II has the same cost formulas as phase I except that the R relation is replaced by the S relation, Rskew replaced by Sskew and t_R replaced by t_S.

Phase III consists of the following steps:

(a) Read R pages for skew bucket and $\left\lceil \frac{B}{P} \right\rceil - 1$ other buckets:

This is the same cost as in Phase I, step (g).

(b) Build an in-memory hash table for each bucket:

$$t_{skewbucket} \cdot (Hash + Move) + \left(\left\lceil \frac{B}{P} \right\rceil - 1 \right)$$
$$\cdot t_{nonskewbucket} \cdot (Hash + Move)$$

(c) Read S pages for skew bucket and $\left\lceil \frac{B}{P} \right\rceil - 1$ other buckets:

This is the same cost as in Phase II, step (g).

(d) Probe the hash table and form the join:

The cost for the non skew buckets is

$$\left(\left\lceil \frac{B}{P} \right\rceil - 1 \right) \cdot \left(\frac{(1 - Sskew) \cdot \{S\}}{B} \right)$$
$$\cdot (Hash + (Comp + Move)$$
$$\cdot \frac{(1 - Rskew) \cdot \{R\}}{JV_R - 1})$$

The cost for the skew bucket is

$$(1 - Fskew) \cdot (Sskew \cdot \{S\} + \frac{(1 - Skew) \cdot \{S\}}{B})$$
$$\cdot (Hash + (Comp + Move)$$
$$\cdot \frac{(1 - Rskew) \cdot \{R\}}{JV_R - 1})$$

$$+ Fskew \cdot (Sskew \cdot \{S\} + \frac{(1 - Skew) \cdot \{S\}}{B})$$
$$\cdot (Hash + (Comp + Move) \cdot Rskew \cdot \{R\})$$

where $Fskew$ is the fraction of tuples in relation S that match the skew value in relation R, i.e.,

$$\frac{Sskew \cdot \{S\}}{Sskew \cdot \{S\} + (1 - Sskew) \cdot \frac{\{S\}}{B}}$$

We now present the analytical cost model for our load balancing hash-join algorithm.

Phase I has the same cost formulas as phase I for the hash-join model.

Phase II has the same cost formulas as phase II for the hash-join model.

Scheduling phase:

$$F \cdot B^2 \cdot Assign$$

where F is a scheduling constant.

Phase III consists of the following steps:

(a) Read R pages for skew bucket and other buckets:

$$IO \cdot [\left\lceil \frac{B-(P-N+1)}{P} \right\rceil \cdot \left\lceil \frac{(1-Rskew) \cdot |R|}{B} \right\rceil + \frac{Rskew \cdot |R| + \left\lceil \frac{(1-Rskew) \cdot |R|}{B} \right\rceil}{N}]$$

where N is the number of processors needed to handle the skew bucket. N is defined as

$$Max(\frac{Rskew \cdot \{R\} + (1-Rskew) \cdot \{R\}/B}{\{R\}/B}, \frac{Sskew \cdot \{S\} + (1-Sskew) \cdot \{S\}/B}{\{S\}/B})$$

The first term gives the number of other buckets that will be handled by the processor that handles the skew bucket times the number of pages for non skew buckets. The second term gives the number of pages per processor for the skew bucket, i.e., number of pages of skew data plus number of pages for non skew data in skew bucket divided by the number of processors that will process the skew data bucket.

(b) Build an in-memory hash table for each bucket:

$$\left\lceil \frac{B-(P-N+1)}{P} \right\rceil \cdot \left\lceil \frac{(1-Rskew) \cdot |R|}{B} \right\rceil \cdot t_R \cdot (Hash + Move)$$

$$+ \frac{Rskew \cdot |R| + \left\lceil \frac{(1-Rskew) \cdot |R|}{B} \right\rceil}{N} \cdot t_R \cdot (Hash + Move)$$

(c) Read S pages for skew bucket and other buckets:

$$IO \cdot [\left\lceil \frac{B-(P-N+1)}{P} \right\rceil \cdot \left\lceil \frac{(1-Sskew) \cdot |S|}{B} \right\rceil + \frac{Sskew \cdot |S| + \left\lceil \frac{(1-Sskew) \cdot |S|}{B} \right\rceil}{N}]$$

(d) Probe the hash table and form the join:

$$\left\lceil \frac{B-(P-N+1)}{P} \right\rceil \cdot \left\lceil \frac{(1-Sskew) \cdot |S|}{B} \right\rceil \cdot t_S \cdot (Hash + (Comp + Move) \cdot \frac{(1 - Rskew) \cdot \{R\}}{JV_R - 1},$$

$$+ N \cdot [(1 - Fskew) \cdot (Sskew \cdot \{S\} + \frac{(1 - Skew) \cdot \{S\}}{B}) \cdot (Hash + (Comp + Move) \cdot \frac{(1 - Rskew) \cdot \{R\}}{JV_R - 1})$$

$$+ Fskew \cdot (Sskew \cdot \{S\} + \frac{(1 - Skew) \cdot \{S\}}{B}) \cdot (Hash + (Comp + Move) \cdot \frac{Rskew \cdot \{R\}}{N})]$$

5. Comparison of Analytical and Experimental Results

In this section we present the results of some of our experiments as well as the results of our analytical models. The hash-join algorithms were executed on a 10 node Sequent Symmetry multiprocessor, in a single user environment. Since the multiprocessor we used does not have a parallel I/O capability, that part of the algorithm had to be simulated. When a page I/O was required, a procedure was called that did a busy wait for the required amount of time, e.g., 24 ms. We ran experiments with a uniform data distribution, single skew in the inner relation, single skew in the outer relation and skew in both relation, i.e., double skew. For the single skew cases, the join selectivity was the same, i.e., 0.0001. For the double skew case, the join selectivity was 0.0406, the result was considerably larger than in the single skew cases. Also, in the double skew case, the CPU processing became the dominating factor in the response time of the algorithms. We also ran experiments that included the cost of writing the resultant relation to secondary storage. We should note that our analytic models were adjusted to accomodate the extra I/O cost, although for brevity we did not show the changes of the cost formulas in the paper. We should note that the data used for the graphs are an average of several runs of the algorithms on the multiprocessor. We should also point out that in our experiments we assumed that enough memory exists to permit an in memory hash table to be built for the maximum bucket size. So for the basic hash-join method, we would not incur bucket overflow. Hence, our experiments deal solely with the load balancing aspect of the hash-join approach in a shared memory multiprocessor.

In figure 1, we show the effect of data skew on the basic hash-join algorithm. An interesting observation concerning the 6 processor case, is that the decrease in response time over the 5 processor case is small. The reason for this, is that for both the 5 and 6 processor cases, a maximum of 5 buckets will be processed during phase III by at least one processor, i.e., the processor that dictates the overall performance. A similar situation occurs for the 8 processor case.

As one would imagine, the response time increases as the data skew increases, as shown in figure 1. For the 2 processor run, the difference between the time taken by the algorithm for a uniform data distribution versus a data skew of 0.4 was only 13.76%. When we used 4 processors the difference increased to about 55.79% and when we used 9 processors the difference reached 97.62%.

In figure 2, we show the effect of data skew on the load balancing hash-join algorithm. Again, as the data skew increases, so does the response time, but not as much as for the basic method. For the 2 processor case, there is a difference of 8.37%, for the 5 processor case there is a difference of 31.85% and for the 9 processor case there is a difference of 52.17%. If we examine the case with skew equal to 0.4, we see that the difference in response time for the basic method and the load balancing method for 2, 5 and 9 processors is 3.93%, 16.96% and 28.75%, respectively. So, we see a reasonable decrease in response time with the load balancing hash-join method.

In figure 3, we compare the basic hash-join method with the load balancing hash-join method for a uniform data distribution. We see that there is little difference between the two. The maximum difference is approximately 1.09%. Hence, the scheduling overhead for the load balancing algorithm is insignificant compared to the entire cost. In addition, we show the results of the models for each of the two algorithms. The maximum difference between the basic method and its model is about 6.97%. The maximum difference between the load balancing method and its model is about 5.43%.

In figure 4, we again compare the basic hash-join algorithm with the load balancing hash-join algorithm but for the case where there is single skew in the inner relation. The skew value of 0.2 indicates that 20% of the tuples in the R relation have the same value. The total percentage of tuples hashing into the skew bucket is 23.2%. For the 8 processor case, the load balancing hash-join algorithm reduces the response time by about 13.83%. The average improvement for 6 to 9 processors is about 10.4%. The results of the models are also shown in figure 4. For the basic algorithm, the maximum difference between the actual run and the model is 3.15%. In the case of the load balancing algorithm and model, the maximum difference is only 8.46%.

For the next set of experiments, we wanted to see the effect of having to write the join result to disk. Writing the resulting relation back to disk dominated the overall cost of the join as can be seen by comparing figure 5 with figure 4. We can

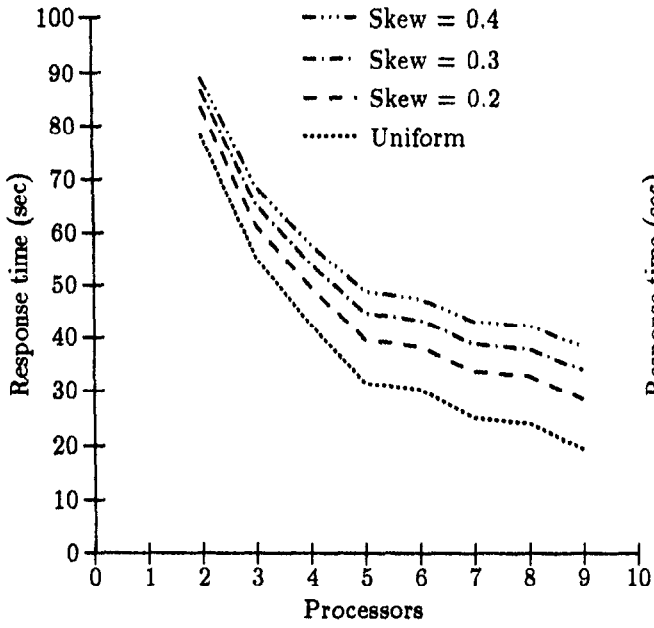

Figure 1: Basic Hash-Join Method

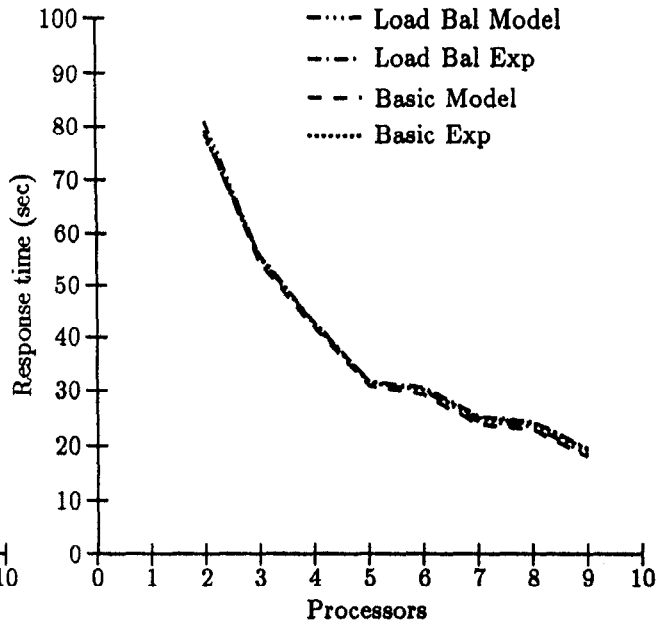

Figure 3: Basic and Load Balancing Hash-Join (uniform data)

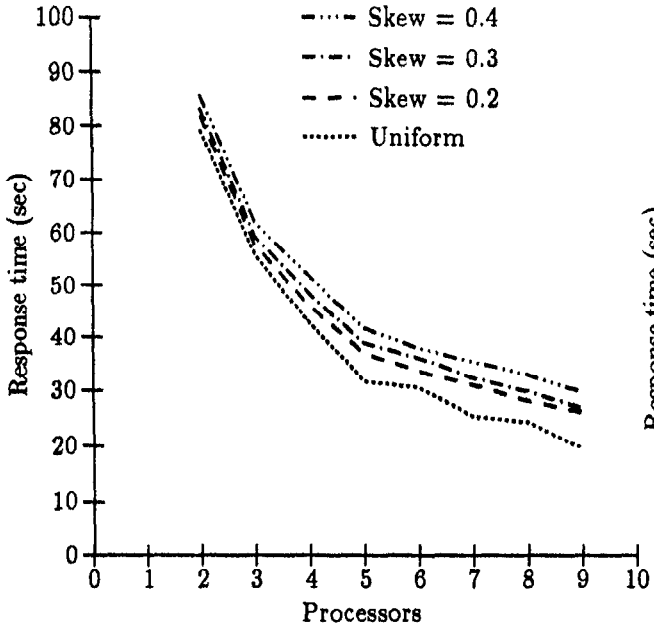

Figure 2: Load Balancing Hash-Join Method

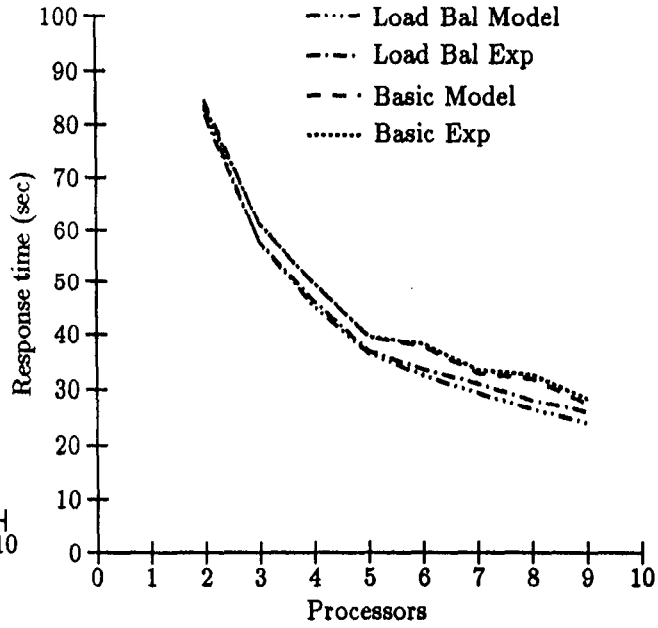

Figure 4: Basic and Load Balancing Hash-Join (R skew = 0.2)

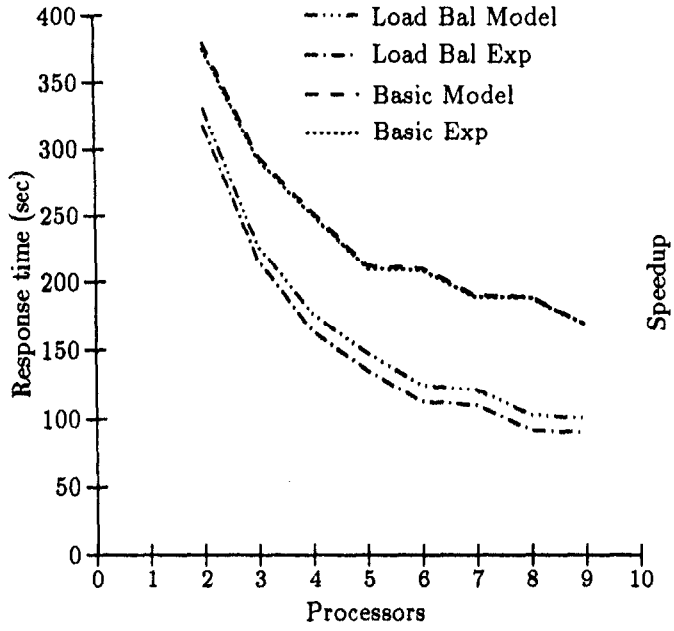

Figure 5: Basic and Load Balancing Hash-Join (R skew = 0.2) with result written to disk

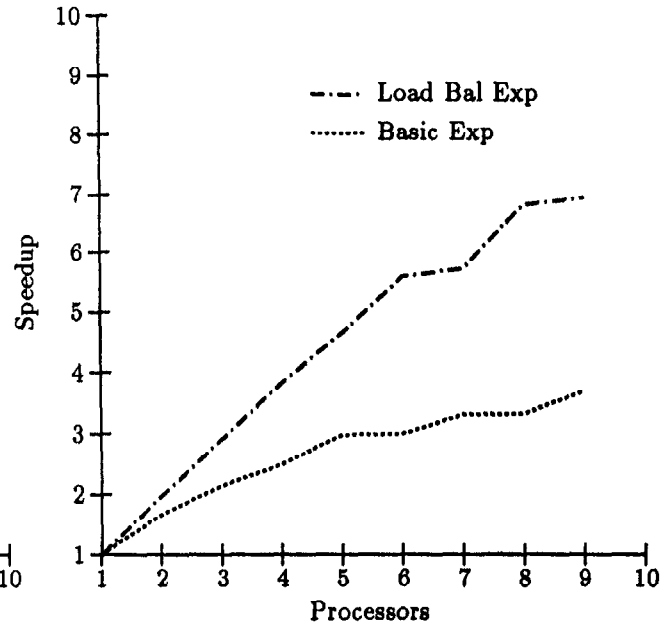

Figure 7: Speedup for Basic and Load Balancing Hash-Join (R skew = 0.2) with result written to disk

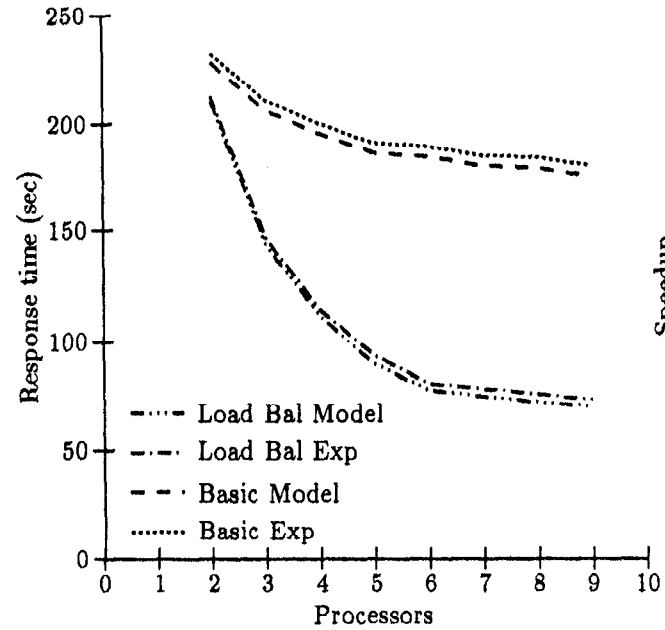

Figure 6: Basic and Load Balancing Hash-Join (R skew = S skew = 0.2)

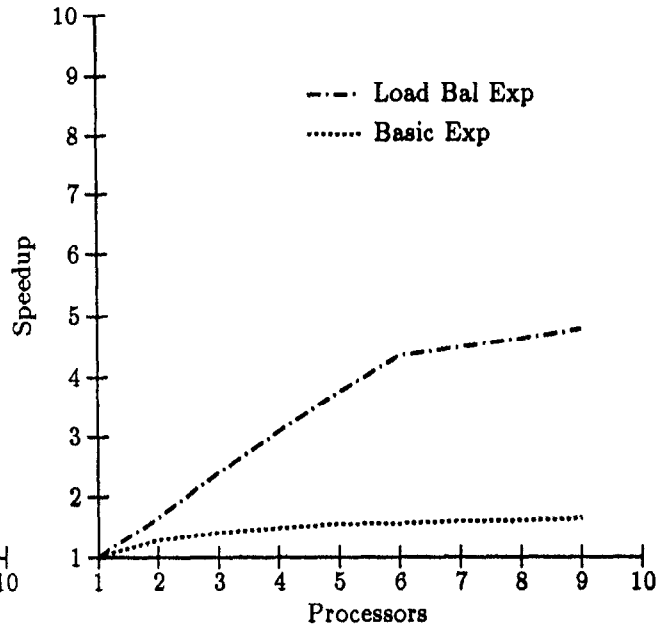

Figure 8: Speedup for Basic and Load Balancing Hash-Join (R skew = S skew = 0.2)

also see in figure 5 that the difference between the two algorithms is even more pronounced. This is due to the fact that in the basic method, one processor writes out the join result from the skewed bucket whereas in the load balancing method multiple processors write out the join result in parallel. In the 2 processor case, the load balancing method shows a decrease of about 15.4% as compared with the basic method. In the 8 processor case, the difference between the two algorithms increases to 51.19%. The average difference for the 2 to 9 processor cases is 33.1%. In addition, the model for both algorithms is quite good. The maximum difference between the observed time and the calculated time for the load balancing case is only 11.83%, while for the basic hash-join case it is less than 1%.

In addition to the response time graph in figure 5, we also include the companion graph, figure 7, of the speedup for the basic hash-join method and the load balancing method. As can be seen in figure 7, the speedup for the basic method is dismal. This can be attributed to the fact that a single processor is responsible for handling the skewed bucket. The small increase in speedup as proccessors are added is probably due to the distribution of some of the I/O costs across the additional disks. As a reminder, the number of disks is set equal to the number of processors. For the load balancing method, the speedup is close to linear, especially up to the 6 processor case. As calculated by the load balancing algorithm, the number of processors needed to handle the skewed bucket, so as to distribute the load evenly, is 6. Hence, the improvement as shown in figure 7. For the 7 processor case, the number of steps needed by the algorithm is the same as for the 6 processor case. A similar situation occurs for the 9 processor and 8 processor cases.

For the final set of experiments to be reported in this paper, we examined the situation of double skew. The results are shown in figure 6. Both the inner and outer relations had a skew value of 0.2. That is, as mentioned previously, 23.2% of the tuples of each relation hashed into one bucket, which was the same bucket, i.e., bucket 0, for each relation. This produced a join selectivity of 0.0406. For the case of double skew as with the case of writing the result to disk, the difference between the performance of the two methods is considerable. The maximum difference occurs for the 9 processor case, i.e., the load balancing algorithm shows a reduction in response time of approximately 59.28%. The average reduction in response time, considering the 2 to 9 processor cases, is 40.7%, with the 2 processor case having the least reduction, i.e., 8.5%. It can also be seen from figure 6 that the cost model is again a good reflection of the algorithm's cost.

In addition to the response time graph in figure 6, we also include the companion graph, figure 8, of the speedup for the basic hash-join method and the load balancing method. As can be seen in figure 8, the speedup for the basic method is virtually nonexistent. This is due to the fact that a single processor is responsible for handling the skewed R and S buckets. The amount of work done by this processor overshadows the total work done by all the other processors. For the load balancing method, we see a reasonable speedup, although not as good as in figure 7. This can be attributed to the fact that the load balancing algorithm calculates the number of needed processors based on the maximum size of the R bucket and S bucket. For the double skew case, the decision should be made based on the combined sizes. Hence, more than 6 processors should be allocated to the skewed bucket.

6. Conclusion

In this paper, we have adapted the Grace hash-join method for a shared everything environment and have designed and implemented a modified version that trys to balance the load on the processors when the data is skewed. We also developed cost models for our algorithms and showed that they accurately reflect the performance of the algorithms, under our assumptions. The algorithms were run on a 10 node Sequent multiprocessor machine with the parallel I/O capability simulated. From our experiments, we saw that even single skew affects the performance of the basic hash-join approach for a shared-everything system. The performance degrades greatly when the result of the join is written to disk or when there is double skew. Our load balancing algorithm has also been shown to have a much better performance when compared with the basic method in all of those cases.

7. Acknowledgments

The author would like to thank the anonymous referees for their insightful and constructive comments.

8. References

[BF87] C. Baru and O. Frieder, "Implementing Realtional Database Operations in a Cube-Connected Multicomputer," Proc. of IEEE Data Engineering Conference, 1987, 36-43.

[DKT90] S. Deen, D. Kannangara and M. Taylor, "Multi-Join on Parallel Processors," IEEE Symposium on Databases in Parallel and Distributed Systems, 1990, 92-102.

[DG85] D. DeWitt and R. Gerber, "Multiprocessor Hash-Based Join Algorithms," Proc. of 11th VLDB Conference, 1985, 151-164.

[DGGHKM86] D. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar and M. Muralikrishna, "GAMMA - A High Performance Dataflow Database Machine," Proc. of VLDB Conference, 1986, 228-237.

[HS78] E. Horowitz and S. Sahni, Fundamentals of Computer Algorithms, Computer Science Press, Potomac, MD, 1978.

[KNHT90] M. Kitsuregawa, M. Nakano, L. Harada and M. Takagi, "Performance Evaluation of Functional Disk System with Nonuniform Data Distribution," Proc. of IEEE Symposium on Databases in Parallel and Distributed Systems, 1990, 80-89.

[KO90] M. Kitsuregawa and Y. Ogawa, "Bucket Spreading Parallel Hash: A New Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer (SDC)," Proc. of 16th VLDB Conference, 1990, 210-221.

[KTM83] M. Kitsuregawa, H. Tanaka and T. Moto-Oka, "Application of Hash to Database Machine and its Architecture," New Generation Computing, 1, 1983, 63-74.

[LY88] S. Lakshima and P. Yu, "Effect of Skew on Join Performance in Parallel Architectures," IEEE Symposium on Databases in Parallel and Distributed Systems, 1988, 107-120.

[LTS90] H. Lu, K. Tan and M. Shan, Hash-Based Join Algorithms for Multiprocessor Computers with Shared Memory," Proc. of 16th VLDB Conference, 1990, 198-209.

[MR89] M. Murphy and D. Rotem, "Effective Resource Utilization for Multiprocessor Join Execution," Proc. of 15th VLDB Conference, 1989, 69-75.

[OL89] E. Omiecinski and E. Lin, "Hash-Based and Index-Based Join Algorithms for Cube and Ring Connected Multicomputers," IEEE Trans. on Knowledge & Data Eng., 1, 3, September 1989, 329-342.

[OS90] E. Omiecinski and R. Shonkwiler, "Parallel Join Processing using Nonclustered Indexes for a Shared Memory Multiprocessor," 2nd IEEE Symposium on Parallel & Distributed Processing, December 1990, pp. 144-151.

[QI88] G. Qadah and K. Irani, "The Join Algorithms on a Shared-Memory Multiprocessor Database Machine," IEEE Trans. on Software Eng., 14, 11, November 1988, 1668-1683.

[SD89] D. Schneider and D. DeWitt, "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment," ACM SIGMOD Proceedings, 1989, 110-122.

[Sha86] L. Shapiro, "Join Processing in Database Systems with Large Main Memories," ACM Trans. on Database Sys., 11, 3, 1986, 239-264.

[SKPO88] M. Stonebraker, R. Katz, D. Patterson and J. Ousterhout, "The Design of XPRS," Proc. of 14th VLDB Conference, 1988, 318-330.

[WDY90] J. Wolf, D. Dias and P. Yu, "An Effective Algorithm for Parallelizing Sort Merge Joins in the Presence of Data Skew," IEEE Symposium on Databases in Parallel and Distributed Systems, 1990, 103-115.