SKEW INSENSITIVE PARALLEL JOIN WITH SAMPLING

Jorge A. Cobb, Shiow-yang Wu and Daniel P. Miranker

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-91-35 November 1991

Skew Insensitive Parallel Join with Sampling

Jorge A. Cobb, Shiow-yang Wu and Daniel P. Miranker

Department of Computer Sciences,
The University of Texas at Austin,
Austin, TX 78712

October 14, 1991

Abstract

The problem of skewed distribution of data values in a relation has been identified as a major performance limiting factor for external parallel join algorithms. The effective parallelism of both sort-merge based and hash based algorithms degrade significantly with skew on the join attribute values as well as skew on the distribution of data among processors. In this paper, we propose an external parallel join algorithm, derived from quick-sort, which performs well even in the case of skew. The algorithm uses range partitioning to split each relation into buckets, where the range values are determined using sampling techniques. Furthermore, each bucket is distributed evenly among all system disks, ensuring that all processors may work in parallel to further partition each bucket. Since both the task of partitioning the relations and the join of the final buckets are evenly distributed over all processors, the problem of skew is solved effectively. Experimental results show that a close to linear speedup can still be obtained even in the case of skew.

1 Introduction

The performance of a database system can be significantly improved if the queries are executed in parallel. In many databases, certain attribute values occur much more frequently than others. This phenomenon is referred to as data skew. Data skew on the joining attribute values can have a severe impact on the performance of a parallel join operation. In [LY89], the performance of a multicomputer with 2 MIPS processors, each with its own disk, is compared analytically against

a 60 MIPS processor with 20 disks. Hash-join was the algorithm of choice for both machines. In the absence of skew, increasing the number of processors in the multicomputer showed a speedup factor of up to 6 with respect to the single large processor. With a skew of only five percent, the multicomputer achieved virtually no speedup. Other studies also show the grave effect of data skew on join performance [LY88a, LY88b, LY89, ID90].

Among the work that has been done on external parallel join algorithms, direct parallelization of the nested-loop join, sort-merge join, and hash join are the most popular (e.g., [Bit82, BBDW83, DG85, ID90, VG84, Men86, ID90]). Parallel nested-loop join is insensitive to skew, but it fails to achieve the desired performance. Parallel sort-merge and hash joins are sensitive to the effect of skew[VG84, DG85].

To overcome these problems, we have developed an external parallel join algorithm motivated by parallel quick-sort algorithms. The algorithm partitions the relations into disjoint joinable buckets based on the range of the values of the joining attribute. Since both range partitioning and hash functions are many-to-one mappings, the algorithm demonstrates the same performance benefits as a hash join. However, for this quick-sort based method the boundaries of the ranges are determined adaptively by sampling the input relations. Any variance in the size of the buckets is balanced by responding adaptively in the next partitioning pass through the data. Except for the first pass, sampling and adjusting the bucket bounds occurs concurrently with the partitioning. Hence, the sampling technique incurs little I/O overhead.

In the presence of skew, even the sampling technique may result in one or more buckets that are much larger than others. If each bucket is stored on a single disk and further partitioned by a single processor, the large buckets would still degrade parallelism by forcing a high level of activity on one disk. To overcome this, each bucket is distributed evenly across all disks.

Many parallel external join algorithms aim to reduce both disk I/O and interprocessor communication. It is our claim that even in a system with multiple disks, either in shared-nothing or a RAID configuration, disk-seek time remains a primary performance bottleneck.[BD88] Consequently, the additional interprocessor communication overhead needed to distribute the buckets does not undermine the total performance of the system.

We have implemented the algorithm on a multicomputer, the Symult 2010. Figure 1 illustrates our abstract machine model. Each processor has its own local memory and disk. All communication between processors is done by message passing. The interconnection network is assumed to be capable of doing parallel transfers between processors and broadcasting messages from any one processor to all other processors. We started with disks on 8 of the processors. The machine is no longer supported. Due to recurrent hardware problems, we present results for a 5 processor machine.

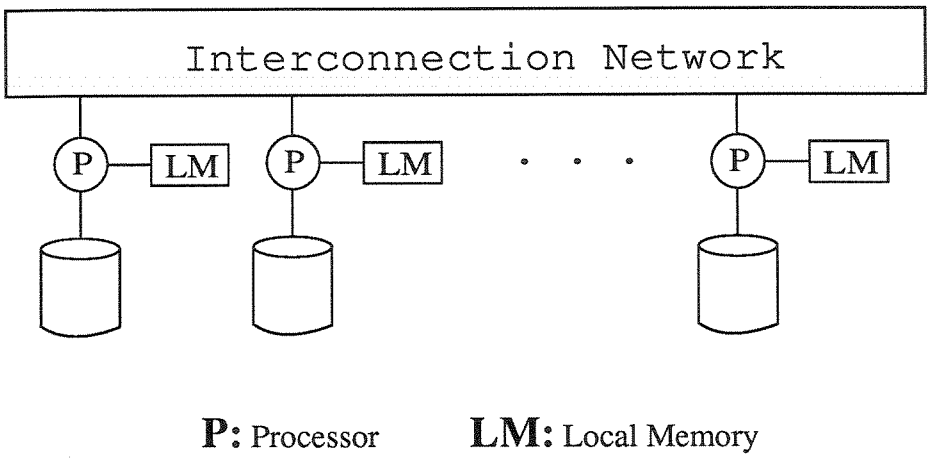

Figure 1: The architectural model.

Our algorithm is general enough to be applied on different architectural models as long as the assumptions presented in this section are reasonably satisfied. The new algorithm is described in Section 2 with a discussion on how the skew is handled effectively. Analytical expressions on the join performance are derived in Section 3. The performance results in Section 4 prove our claim that the new algorithm is both efficient and effective in handling skew. In Section 5, our conclusions and plans for further research are presented.

2 Join Algorithm

Let R and S be the relations to be joined. Without loss of generality, we assume that R is smaller than S. R and S are called the *outer* and *inner* relations, respectively. Both relations are assumed to be distributed horizontally and evenly among all disks.

The algorithm begins by partitioning relation R into disjoint buckets, each of which should be small enough to fit in main memory. If we have a total of B output buffers, then we can only split R into B buckets. If R is large enough, some or all of these buckets will be too large to fit in main memory. Thus, we recursively partition each of these large buckets into B sub-buckets. That is, we construct a tree of fanout B, where the children of a node are its B disjoint partitions. The root of the tree is relation R, and the leaves of the tree are a disjoint partitioning of relation R, where each leaf is small enough to fit in main memory.

We use *range partitioning* to split relation R into buckets. That is, each bucket has a lower

and upper bound on the joining attribute value that corresponds to the bucket. Each tuple in the relation is sent to the bucket where its joining attribute value is within the bounds set by the bucket. The bounds on the attribute values of each bucket should be such that all buckets will have approximately the same number of tuples. Since we are unaware of the distribution of attribute values in the relation, we obtain a sample of the attribute values in the relation to guide us in choosing a good range of values for each bucket.

To partition the outer relation R, we require a sample of the joining attribute values in the relation. Since relation R is distributed over all disks, all processors sample the relation in parallel and send their sample to a single processor. This processor sorts the samples collected and chooses $B-1$ split values from the sample. If the sample is of size l, every (l/B)th value in the sorted sample is chosen as a split value. The $B-1$ values are then broadcasted to all processors.

Upon receiving the split values, each processors splits the fragment of R stored on its disk into B buckets. A tuple is sent to bucket i, $2 \leq i \leq B-1$, if its joining attribute value is larger than the $(i-1)$th split value but less than or equal to the ith split value. Bucket 1 receives those tuples less than all split values, and bucket B receives those tuples that are larger than all split values.

In the presence of skew, a few buckets may be much larger than others. If each of these buckets were stored in a single disk, then storing and retrieving these buckets would place undue strain on a few disks. To avoid this, and taking advantage of our fast interconnection network, the tuples for each bucket are distributed among all disks. That is, as each processor partitions relation R, it distributes the tuples among the other processors in a round-robin fashion. This will allow the fast retrieval and storage of large buckets.

Let P be the number of processors. Since each of the B buckets could be large enough to require further splitting, each processor keeps a sample of the tuples that it has sent to each bucket. After the partitioning of R terminates, for each bucket b, $1 \leq b \leq B$, processor $b \bmod P$ collects the samples of bucket b from all other processors, sorts the samples, finds the $B-1$ split values for the bucket, and broadcasts these values to all processors. In this way, all processors will be informed of the split values of each bucket.

Once the partitioning values for all B buckets are obtained, each bucket requiring further partitioning will be recursively partitioned in parallel by all processors. That is, if bucket 1 needs further partitioning, all processors will begin to partition it. Once the partitioning of bucket 1 completes, all processors begin to partition bucket 2 in parallel, etc. . A bucket will not be partitioned any further if it is small enough to fit in main memory or if the sampling indicates that the bucket consists primarily of a single value.

To partition relation S, we must ensure that the final buckets of S have the same range of values as their corresponding buckets of R. During the partitioning of outer relation R, a list is kept of

each final bucket of R, along with the upper and lower bounds on the attribute values used to create the bucket. A tree of fan-out B is constructed, where the root represents the inner relation S, and the leaf nodes are a list of final buckets with the same upper and lower attribute value bounds as the final buckets of R. The values used to split each interior node are obtained from the upper and lower bounds on the attribute values of the leaf nodes that are its descendants.

After both relations are partitioned, a list of all the final buckets of both relations along with the size of each bucket is collected and maintained by a coordinator process. Since in the presence of skew a bucket may be so large as to not fit in a processor's memory, the join of a pair of buckets is done using blocked nested loops, where a block is the amount of tuples that fit in main memory. For example, let A and B be a pair of buckets to be joined. If bucket A fits in main memory but bucket B is twice the allowed maximum size, then one processor must join bucket A with the first half of bucket B, and a second processor must join bucket A with the second half of bucket B. In the absence of skew, each bucket will never be larger than a single block.

Each processor repeatedly asks the coordinator for a pair of corresponding buckets to join and the block of each that must be joined. This continues until all pairs of buckets are joined. The result of the join of each pair of buckets is again distributed across all disks. The join of a bucket pair can be done by any efficient internal join algorithm. Our current implementation joins a pair of buckets using sort-merge join, where the sort algorithm is heap-sort.

A pseudo-code description of the algorithm is given in appendix A.

In partitioning the outer relation, the sampling algorithm **S** [Vit84] is used. The description of this sampling algorithm is given in appendix B. This algorithm requires the size of the input relation before it executes. This is of no concern since the relation size is known prior to the execution of the algorithm. Furthermore, the outer relation does not need to be scanned to find the desired tuples. When a tuple is selected for the sample, a seek operation on the file is performed to place the file pointer at the location of the selected tuple. Hence, the total number of I/O operations is proportional to the sample size. In our experimental results, it took less than seven seconds to sample a relation with 250-K tuples.

As each bucket is being formed, each node samples the tuples that it generates for that bucket for possible further partitioning. Since the size of the bucket is not known in advance, the **R** reservoir algorithm [Vit85] is used. The description of this sampling algorithm is given in appendix C. Since the samples are obtained from the tuples generated by the partitioning of a relation, no extra I/O overhead is incurred.

An important issue here is the way we distribute tuples over all disks such that the load is automatically balanced. When each node reads its fragment of the relation and generates tuples for each bucket, the tuples are distributed among all disks. A structure for keeping bucket information

is maintained in each node. When a bucket needs to be split further, all nodes will work on the same bucket in parallel. This technique significantly reduces the among of message passing that would be needed if the bucket information were maintained centralized in one node. Since the job of collecting the samples and finding the partitioning values for each bucket is distributed across all nodes, a high processor utilization and load balancing is effectively achieved.

3 Complexity Analysis

We now compute the disk I/O costs and interprocessor communication cost of the algorithm. We will conclude that the interprocessor communication cost is small compared to disk I/O. Consequently, the additional communication steps used to redistribute the data are worthwhile.

Analysis Parameters

For the purpose of illustration, we will assume the following set of parameters on the multiprocessor machine and the relations to be joined. The join operation is performed on relations R and S, where the size of R is smaller than or equal to the size of S. The join result is denoted by T. Data are assumed to be processed in units of a page.

B: Number of buckets for partitioning the original relations or recursively partitioning the buckets which are still larger than the local memory size. The partitioning tree has, therefore, a fan-out of B in all interior nodes. B is chosen to be as large as the system permits. The limit is usually imposed by the maximal number of files that can be opened at the same time.

m: Number of pages in R.

n: Number of pages in S, $m \leq n$.

FR: Filtering factor[1] of relation R.

FS: Filtering factor of relation S.

t: Number of pages in T. The actual value of t depends on the join relations. However, it is always true that $t \leq m \cdot FR \times n \cdot FS$.

P: Number of processor-disk pairs.

[1]The *filtering factor* of a relation in a join operation is the fraction of tuples that actually participate in the join.

L: The size of half of the local memory in each processor in units of page frames, where a page frame is of the same size as a disk page. The size of each final bucket should not exceed L. This is to ensure that at the join phase, each processor can read in two whole buckets, one from each relation, and join them. We assume that an additional page frame is available on each processor as a buffer for result tuples of the join operation. Therefore, the actual size of local memory is $2L + 1$.

For analyzing the performance of the algorithm, the following cost parameters are assumed. Since the algorithm distributes intermediate results over all disks, the cost of local and remote access must be considered separately. And, no analysis can be complete without the measurement of communication cost.

C_r: The average cost of reading a page from local disk.

C_w: The average cost of writing a page to local disk.

C_{dm}: The average cost of sending a page through the network.

I/O Time

In an algorithm for external join, the dominating cost is usually the time spent on I/O. In our algorithm, for each additional level of partitioning, the whole relation needs to be read from and written back to disks. Since the work is distributed over all processors, each processor only needs to process $1/P$ of the pages. For partitioning R, the partitioning tree is in general a complete tree, because the sampling algorithms are quite effective in splitting the relation into buckets of approximately equal size. The number of levels is therefore $\log_B(m/L)$. Since all processors are operated in parallel, the I/O cost for partitioning R is

$$\frac{m}{P} \cdot \log_B \frac{m}{L} \cdot (C_r + C_w).$$

Similarly, the cost of partitioning S is

$$\frac{n}{P} \cdot \log_B \frac{n}{L} \cdot (C_r + C_w).$$

Finally, we need to read and actually join the corresponding buckets, then write the result relation back to disk. The I/O cost for this operation is

$$\frac{t}{P} \cdot (C_r + C_w).$$

The total cost for I/O is then the sum of the three above:

$$\frac{m}{P} \cdot \log_B \frac{m}{L} \cdot (C_r + C_w) + \frac{n}{P} \cdot \log_B \frac{n}{L} \cdot (C_r + C_w) + \frac{t}{P} \cdot (C_r + C_w).$$

This can be simplified to

$$C_{IO} = \frac{1}{P}(m \cdot \log_B \frac{m}{L} + n \cdot \log_B \frac{n}{L} + t) \cdot (C_r + C_w).$$

It is important to note that the result above is insensitive to data skew either on the join attribute values or the distribution of data. This is the result of our use of sampling techniques and the even distribution of intermediate results. From the formula, it is also true that the value of B should be as large as possible.

Communication Cost

One of the major concerns on the performance of our algorithm is the cost of message passing since all intermediate results are evenly distributed over all disks. In this section, we show that the communication cost is small in comparison with the I/O time and, therefore, it is cost worthy to distribute the intermediate results.

Two types of communication are needed in our algorithm. The first type is the sending and receiving of samples to determine the partition values. These messages are called *sampling messages*. The other type is the transfer of data blocks between processors to distribute the buckets. We call these messages *data messages*. In general, sampling messages are much shorter than data messages. The cost of sampling messages can therefore be safely ignored. In the following analysis, we first compute the total number of messages sent over the network. The cost for communication is then the product of the result with the average cost of sending a page through the network.

At each level of the partitioning tree, each processor processes m/P pages and distributes a portion of $(P-1)/P$ of it to other processors. The number of messages sent for partitioning R is

$$\frac{m(P-1)}{P^2} \cdot \log_B \frac{m}{L}.$$

The partition of S takes

$$\frac{n(P-1)}{P^2} \cdot \log_B \frac{n}{L}$$

messages. Finally, to distribute the result,

$$\frac{t(P-1)}{P^2}$$

messages are required. The numbers above are for one processor. The total number of messages sent by P processors is therefore

$$\frac{P-1}{P}(m \cdot \log_B \frac{m}{L} + n \cdot \log_B \frac{n}{L} + t).$$

The communication cost is thus

$$\frac{P-1}{P}(m \cdot \log_B \frac{m}{L} + n \cdot \log_B \frac{n}{L} + t) \cdot C_{dm}.$$

Comparing this cost with C_{IO}, we can see that the distinguishing factors are the cost of reading/writing a page on a local disk vs. the network throughput. If $(P-1) \cdot C_{dm} \ll (C_r + C_w)$, then the approach of distributing intermediate results over all disks to avoid the skew problem would be cost worthy. The Symult network can transfer messages at a rate of 20 megabytes per second between a pair of nodes. Since several messages can be transmitted in parallel, even a small 6 by 6 mesh could reach a throughput of more than 100 megabytes per second. This gives us a C_{dm} cost of about 0.01 microseconds/byte. The cost $(C_r + C_w)$ for most disks is at least 2 microsecond/byte. [2] Each processor would require several disks before the network becomes the bottleneck of the system. Hence, distributing intermediate results over all disks does not hinder the performance of the algorithm.

4 Performance Results

4.1 The Implementation Machine

The join algorithm was implemented on a Symult 2010 multiprocessor. The Symult 2010 used for this experiment consists of 24 nodes. Each of these nodes is equipped with its own processor, memory, and communications interface. Eight of these nodes have a disk attached to them. Due to hardware problems with three of these disks, we present results for a five disk system.

Communication between nodes is accomplished through message passing. The communications network consists of a mesh of specialized VLSI routing chips. The network uses worm-hole routing and is synchronous, which allows for transmission rates of above 20 megabytes per second.

The memory capacity of the diskless nodes is one megabyte. More than half of this memory is consumed by the operating system kernel residing in the node. Each of the five disk nodes has a memory capacity of three megabytes. Two thirds of the memory in the disk node is occupied by the kernel[3], leaving about one megabyte of user space. Due to the memory limitations of the diskless nodes, the algorithm was designed to run only in the disk nodes.

[2] We note that these ratios continue to hold when considering RAID disks in conjunction with present generation parallel computers as well as comparing disk latency and interprocessor communication latency.

[3] The kernel in a disk node is much larger that the kernel in a diskless node because the disk node requires all the file handling routines.

The fanout B for the algorithm was set to 19, since a processor can only have 20 files open at once. That is, one file descriptor is used for the bucket being split and the other 19 are used for the sub-buckets being created. The maximum size allowed for a bucket is 1,500 tuples, since no more that 3,000 tuples fit in main memory.

4.2 The Test Database

All relations consists of tuples with a length of 200 bytes. The joining attribute is a positive four-byte integer. The remaining bytes of the tuple are left blank. The tuples of each relation have been randomly distributed across all disks. Furthermore, a different seed for the random number generator is used when creating each relation.

The relation sizes vary from 50-K tuples up to 250-K tuples in increments of 50-K tuples. Due to the fanout B and the maximum bucket size given above, the height of the split tree of each relation was 2. That is, each relation was split into B buckets, and each of these was split into B sub-buckets.

Every outer relation is joined with an inner relation that contains the same number of tuples as the outer relation. The join result will also contain the same number of tuples as the inner and outer relations. No projection is done on the attributes of the join result, which gives a result tuple size of 400 bytes.

Three different distribution schemes on the joining attribute values are used: uniform, non-uniform, and skewed.

4.3 Uniform and Non-uniform Distribution

For the uniform distribution of values, the relations have joining attribute values in the range:

$$0 \ldots Cardinality - 1.$$

That is, no duplicate joining attribute values are allowed. Each uniform outer relation is joined with a uniform inner relation of the same cardinality.

The non-uniform distribution was chosen to test the effectiveness of sampling. In this distribution, a large number of tuples are given a joining attribute value in a small region of the total range of values of the relation. If the sampling is ineffective, some buckets will become much larger than others. This in turn will cause further bucket splits than necessary, decreasing performance.

The relations for this case have joining attribute values in the range:

$$0 \ldots 2 * Cardinality - 1.$$

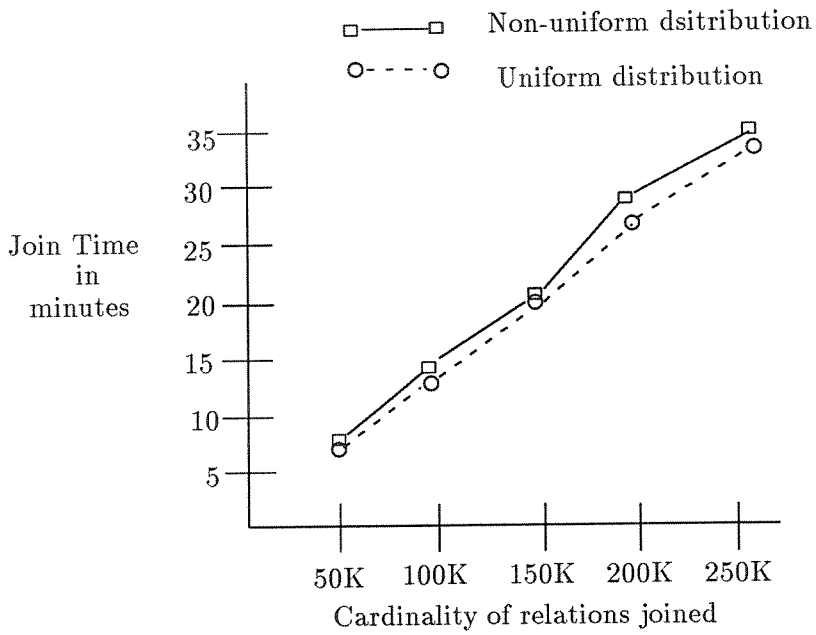

Figure 2: Uniform vs. non-uniform distribution join results

One half of the tuples have attribute values in the range:

$$0 \ldots Cardinality/2 - 1,$$

and the other half are in the range:

$$Cardinality/2 \ldots 2 * Cardinality - 1.$$

No duplicate joining attribute values are allowed. Each non-uniform outer relation is joined with a non-uniform inner relation of the same cardinality.

Figure 2 shows the results of the uniform distribution joins and the non-uniform distribution joins. As desired, the join time increases linearly with the size of the relations. It must be noted, however, that if the relations reached a cardinality large enough to add a third level to the split tree, then a "step" in the join time function would be noticed at that point. This would probably happen at around 400K tuple relations. Due to limitations in our disk capacity, we were not able to perform joins of such large relations.

The sampling techniques were not affected by the non-uniform distribution and yielded buckets of similar size. Thus, the non-uniform distribution joins had similar joining times as the uniform distribution joins.

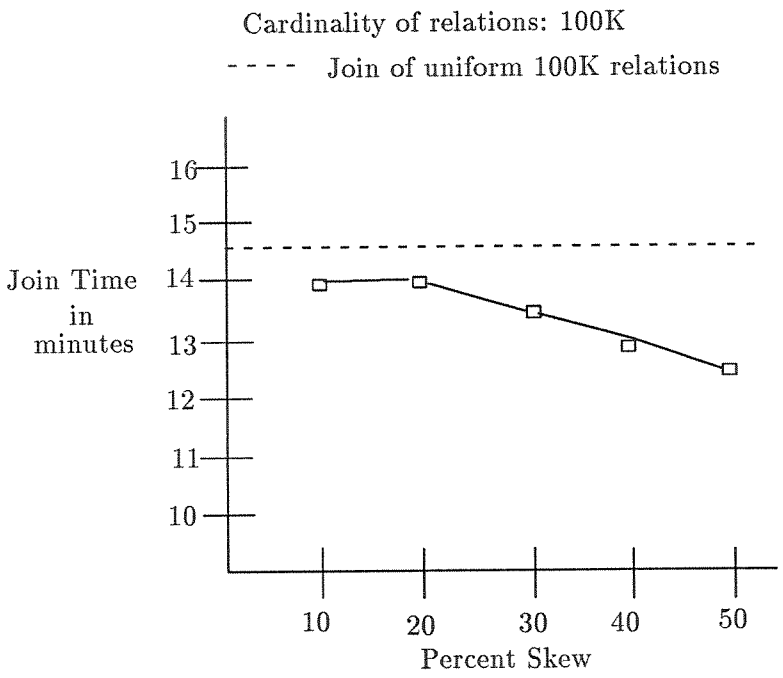

Figure 3: Skewed distribution join results

4.4 Skewed Distribution

The outer relation for this case has a uniform distribution of joining attribute values.

The inner relation has a fraction of its tuples with the same joining attribute value. If this fraction is X, $0 \leq X \leq 1.0$, then a total of $X * Cardinality$ tuples have a joining attribute value of $Cardinality - 1$. The remainder of the tuples have a unique value in the range:

$$0 \ldots Cardinality * (1.0 - X).$$

Thus, the inner, outer, and result relations have the same cardinality. The relation size is kept constant at 100-K tuples. The skew percentage is varied from 10% to 50% in increments of 10%.

Figure 3 shows the results for the skew distribution joins. It is also shown in the figure the join time of two uniform 100-K tuple relations. The join time for the skew join is actually less than the uniform join time. This is caused by the reduced amount of I/O needed in the skew join. Although the result relation in the skew join is of the same size as in the uniform case, less I/O is needed to create the result in the skew case. That is, many buckets of the outer relation have corresponding buckets in the inner relation that are empty. During the joining of buckets phase, these outer buckets do not need to be joined.

5 Conclusions

The parallel join of two relations typically consists of two phases. First, the joining relations are split into buckets of similar size. Second, the corresponding pairs of buckets are joined in parallel by assigning the join of each bucket pair to a processor.

To perform the first step, we chose to use sampling techniques. These have the advantage over hashing techniques that the values used to split the relation originate from the relations themselves, rather than relying on a hashing function which in some cases may lead to a high degree of collision.

In the presence of skew, one or more buckets may be much larger than the others due to the multiple occurrences of a single value. If each of these buckets were stored in a single disk, it would degrade parallelism by forcing a high level of activity on one disk. To overcome this problem, each bucket is uniformly distributed over all disks. This balances the work load among all disks, eliminating the performance degradation caused by skewed data.

As long as the network is fast enough, distributing tuples over all disks should not degrade performance. The Symult network has a bandwidth of about 20 megabytes/second for a single message. Since the network is configured as a mesh, several messages can be transmitted in parallel, leading to an effective bandwidth of hundreds of megabytes per second. To flood this network, it would require a very large number of disks.

The experimental results show that the algorithm's running time is approximately linear with respect to relation size. Furthermore, the presence of skew does not degrade the performance as compared to a skew-free distribution.

This work was conducted as part of the DATEX expert-database system project. Large rule-based applications contain very long multiway joins. Our future includes modifying the algorithm to sample and partition both relations at once, rather than partitioning the outer relation first followed by the partitioning of the inner relation. Also, we plan to perform tests using double skew, that is, when both the outer and inner relations have a skewed distribution of values.

References

[Bat68] Batcher, K.E., "Sorting networks and their applications," *Proc. AFIPS Spring Jt. Computer Conf.*, Vol. 32, AFIPS Press, Arlington, Va.

[Bit82] Bitton-Friedland, D., Design, Analysis and Implementation of Parallel External Sorting Algorithms, Ph.D. dissertation, TR464, Computer Science Department, Univ. of Wisconsin, Madison, January 1982.

[BBDW83] Bitton, D., H. Boral, D.J. DeWitt, and W.K. Wilkinson, "Parallel Algorithms for Relational Database Operations," *ACM Trans. Database Systems*, Vol. 8, No. 3, Sep. 1983, 324-353.

[BD88] Boral H. and D.J. DeWitt, "Database Machines: An Idea Whose Time has Passed", reprinted in: *Tutorial: Parallel Architectures for Database Systems*, A.R. Hurson, L.L. Miller and S.H. Pakzad, eds., IEEE Computer Society Press, 1988.

[DG85] DeWitt, D.J. and R.H. Gerber, "Multiprocessor Hash-based Join Algorithms," in *Proc. Intl. Conf. on Very Large Data Bases*, 1985, 151-163.

[ID90] Iyer, B.R. and D.M. Dias, "System Issues in Parallel Sorting for Database Systems," *Proc. Intl. Conf. Data Engineering*, 1990, 246-255.

[LY88a] Lakshmi, M.S. and P.S. Yu, "Effect of Skew on Join Performance in Parallel Architectures," *Intl. Symp. Database in Parallel & Distributed Systems*, 1988, 107-117.

[LY88b] Lakshmi, M.S. and P.S. Yu, "Performance of Relational Join Operations on Parallel Architectures," *IBM Research Report RC 13370*, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, 1988.

[LY89] Lakshmi, M.S. and P.S. Yu, "Limiting Factors of Join Performance on Parallel Processors," *Proc. Intl. Conf. Data Engineering*, 1989.

[Men86] Menon, J., "A Study of Sort Algorithms for Multiprocessor Database Machines," *Proc. Intl. Conf. on Very Large Data Bases*, 1986, 197-206.

[VG84] Valduriez, P. and G. Gardarin, "Join and Semijoin Algorithms for a Multiprocessor Database Machine," *ACM Trans. Database Systems*, Vol. 9, No. 1, March 1984, 133-161.

[Vit84] Vitter, J.S. "Fast Methods for Random Sampling", *Comm ACM*, Vol. 27, No. 7, July 1984.

[Vit85] Vitter, J.S. "Random Sampling with a Reservoir", *ACM Trans. Mathematical Software*, Vol. 11, No. 1, March 1985.

Appendix

A Join Algorithm Pseudo Code

```
                              JOIN PROCESS
notation: l[i], i>0, is the ith element of list l, P is the number of
          processors, B is the tree fanout, and mynode() returns the processor
          number 0..P-1.
outer_rel_final_buckets - list of integer pairs (less_than_or_eq, size)

begin join
  sample outer relation using S algorithm.
  sort the sample and find the B-1 split values.
  call split_outer_rel(outer_relation, (B-1 split values, MAXINT) ).
  call split_inner_rel(inner_relation, outer_rel_final_buckets).
  call join_buckets().
end join

begin split_outer_rel(file_to_split, file_split_values)
   parameters:
     file_to_split - file to split
     file_split_values - list of B integers

   while not end_of_file(file_to_split)
      read the next tuple
      using file_split_values, find the bucket corresponding to the tuple,
         write the tuple to the tuple's bucket, and
         sample the tuple for the tuple's bucket using algorithm R.

   for each bucket 1..B
      send the collected sample of the bucket to node (B mod P).

   for each bucket 1..B such that (bucket mod P) = mynode()
      receive sample for bucket from all nodes along with
         the size of the bucket stored at the sending node's disk.
```

```
      sort the total sample of bucket and find the B-1 split values.
      send to all nodes the B-1 split values found and the bucket's total size.

   for each bucket 1..B
      receive the B-1 split values and the bucket size.

   for each bucket 1..B
      if bucket size fits in memory or at least 2/3 of the split values
         received for the bucket are identical,
      then  if bucket size is not 0 then
               add to outer_rel_final_buckets the pair
                                    (file_split_values[bucket], bucket size)
      else
        split_outer_rel( bucket file,
                         (B-1 split values received, file_split_values[bucket])
                       )
end split_outer_rel.

begin split_inner_rel(file_to_split, bucket_list)
   parameters:
     file_to_split - file to split.
     bucket_list -
         list of integer pairs (less_than_or_eq, size).

   if the list is of length 1 then return.

   split the bucket_list into B equal sized disjoin sublists,
      name the sublists sublist_1 through sublist_B.

   let file_split_values[i] 1<=i<=B be the less_than_or_equal item
                        of the last element of sublist_i.

   while not end_of_file(file_to_split)
      read the next tuple.
      write tuple to corresponding bucket according to file_split_values.
```

```
   for each bucket 1..B of the file_to_split
      split_inner_rel(bucket file, sublist_bucket).
end split_inner_rel

begin join_buckets()
<L1>
   request from the coordinator a (bucket, block) of the outer relation
   and a (bucket, block) of the inner relation that must be joined.

   if no more buckets to join then return.

   join the block of the outer relation with the block of the inner
      relation using sort merge and write the result

   goto <L1>
end join_buckets
```

COORDINATOR PROCESS

```
begin coordinator
   for each (outer bucket, inner bucket) pair that needs to be joined
      for each block i of the outer relation
         for each block j of the inner relation

            send (outer bucket, i) and (inner bucket, j) to the next node
            that requests a pair of buckets to join.

   inform each node that there are no more buckets to join.
end coordinator
```

B Sampling Algorithm For Outer Relation

Algorithm S

This method sequentially selects n records

```
            at random from a file containing N records.

Step 1 - Generate a random variate U that is uniformly distributed
         between 0 and 1.
Step 2 - If N*U > n then goto Step 4.
Step 3 - Select the next record in the file for the sample.
         Let n := n-1.
         Let N := N-1.
         If n>0 then  goto Step 1 else exit
Step 4 - Skip over the next record (* do not include it in the sample *).
         N := N - 1.
         goto Step 1.
```

C Reservoir Sampling Algorithm For Buckets

Algorithm R

Algorithm R obtains a sample of size n from a record stream. READ_NEXT_RECORD() returns the next record in the stream. C[0..n-1] is an array of records.

```
for j:=0 to n-1                      (* make the first n records candidates *)
    C[j] := READ_NEXT_RECORD();      (* for the sample *)
t:=n;                                (* t = number of records processed so far *)
while not end-of-file
    t := t + 1;
    M := TRUNC(t*RANDOM());          (* M is random in the range 0 <= M <= t-1 *)
    if M < n then
       C[M] := READ_NEXT_RECORD(); (* Make the record a candidate, replacing *)
                                     (* the previous one *)
    else  READ_NEXT_RECORD();        (* skip the record *)
```