

DBJ – A Dynamic Balancing Hash Join Algorithm in Multiprocessor Database Systems (Extended Abstract)

X. Zhao, R.G. Johnson, N.J. Martin

Department of Computer Science
Birkbeck College, University of London
London WC1E 7HX, UK

1 Introduction

Hash join algorithms are a popular query processing mechanism in parallel database systems. A number of papers have addressed the implementation of parallel hash join algorithms including [1], [2], [3], [4], [5], [6], [7], and [8]. Since selections or projections are often performed in advance of the later join operations in order to reduce data volumes, it is difficult to predict the distribution of input relations to a join operation. No hash function can guarantee a uniform output distribution without knowing the input distribution information [9]. The non-uniform output distribution of hash functions causes unbalanced workloads between processors. These unbalanced workloads can curtail scalability and degrade the performance. To take full advantage of parallel processing for a hash-based join algorithm, a workload balancing mechanism is necessary.

Data skew leading to unbalanced workloads may arise from several causes as noted by Walton et al. [8]. A number of skew-handling hash join algorithms have been proposed, including those of Kitsuregawa et al. [4], Hua et al. [3], Wolf et al. [10], and Dewitt et al. [2]. Most of them rely on obtaining knowledge of the source data distributions in order to reduce the unbalance arising from the join processing stages. The algorithms based on preprocessing to obtain the source data distribution result in a high cost. The DBJ algorithm presented needs no preprocessing and relies on no distribution assumptions. It detects the actual hash function output and corrects any unbalanced output dynamically. This is achieved without the need for a co-ordinating processor. This algorithm has been fully implemented in a multiprocessor database system.

2 The DBJ Algorithm

The DBJ algorithm is an architecture independent algorithm. We only assume in the following that certain processors, referred to as data servers, have directly attached disk drives and are used primarily for caching and organizing source relation data. Certain processors, referred to simply as processors, are used for

the actual query processing. In fact a single physical processing component may act as both data server and processor.

The essence of hashing schemes is to use the key value of a tuple to compute the location for the further processing of that tuple. Let R and S be the two relations to be joined. These relations are assumed to be horizontally partitioned and uniformly distributed across all the data servers. Let R_d and S_d be the portion of R and S residing at the data server d and $|R_d|$ and $|S_d|$ be the cardinalities of each portion. Let $D + 1$ be the number of data servers with the identifiers of the data servers being $0, 1, \dots, D$. A hash function h partitions each R_d into $B + 1$ number of buckets, $R_d^0, R_d^1, \dots, R_d^B$. The identifiers of buckets are $0, 1, \dots, B$. Suppose there are $P + 1$ processors and their identifiers are $0, 1, \dots, P$. Suppose at a timepoint the data server d has partitioned Q_d tuples and $q = Q_d/|R_d|$. Let C_{db}^{qp} be the number of tuples in bucket b which is allocated to processor p on the data server d at the moment when the ratio q of R_d 's tuples has been partitioned. Each bucket is allocated to one processor and each processor may be allocated many buckets. Let function f map bucket identifiers to processor identifiers and $F(x)$ represent the set of the bucket identifiers with the same processor identifier x . The value C_{db}^{qp} gives the data distribution information during a partition phase.

On average there will be α buckets on each processor, where $\alpha = \lfloor (B + 1)/(P + 1) \rfloor$. The data servers initially allocate an equal number of buckets of R to each processor. The buckets, $R_d^{n(P+1)+p}$, $n = 0, 1, \dots, \alpha$, are allocated to processor p initially by data server d , $d = 0, 1, \dots, D$. At a timepoint before the partition phase finishes, each data server d has its own distribution information of $C_{dj}^{f(j)}$, $j = 0, 1, \dots, B$. Each data server broadcasts this information to all other servers and the processors, thus allowing the entire current bucket distribution pattern, $C_j^{f(j)} = \sum_{i=0}^D C_{ij}^{f(j)}$, $j = 0, 1, \dots, B$, to be organised on all the data servers and processors. The $\sum$ signifies that all C values with the same bucket identifier are added together. This distribution information also can be seen as a measure of the processor workloads, $C^k = \sum_{j \in F(k)} C_j^k$, $k = 0, 1, \dots, P$. Since every data server and processor knows the distribution information, the balance check can take place on each of them independently and the balancing process can begin when it is necessary.

3 Dynamic Balance

The portion R_d on each data server is partitioned into $R_d^0, R_d^1, \dots, R_d^B$ and moved to the $P + 1$ processors. On each processor the buckets coming from different data servers are composed into complete single buckets, $R^j = \sum_{i=0}^D R_i^j$, $j = 0, 1, \dots, B$. With the initial allocation of buckets, $B + 1$ bucket buffers on each data server are assigned the processor identifiers, and each tuple is put into one of the buffers on the basis of its hashed key value. When the buffers are full, they are sent to the respective processors and the buffers are cleared for further use. Each data server d maintains the local counts $C_{dj}^{qf(j)}$, $j = 0, 1, \dots, B$ and receives the

counts $C_{ij}^{qf(j)}$, $i = 0, 1, \dots, D$ where $i \neq d$, $j = 0, 1, \dots, B$, from the other data servers. The processors receive all counts, $C_{ij}^{qf(j)}$ where $i = 0, 1, \dots, D$ and $j = 0, 1, \dots, B$. The overall distribution, $C^{qk} = \sum_{j \in F(k)} \sum_{i=0}^D C_{ij}^{qk}$, $k = 0, 1, \dots, P$, is formed on all data servers and processors by broadcasting the counts from every data server.

A value β_d can be used to measure the progress of the partition phase on data server d . β_d represents the ratio of the amount of data that has been partitioned on data server d to the total amount of data that is going to be partitioned on d . Hence, β_d gives the percentage of the partition phase completed on each data server. When β_d reaches particular threshold values for the first time during the partition phase, a ‘checkpoint’ occurs and the data server d sends the bucket count information to the other servers and processors. Threshold values can be, for example, 10%, 20%, ..., 100%. At a checkpoint each data server has finished approximately the same amount of work. By basing the distribution of workload information on threshold values at each data server, it ensures that the workloads on each processor can be examined at the same point of the partition process. At a checkpoint, all β_d are the same, q approximately equals β_d , and C^{qk} , $k = 0, 1, \dots, P$, gives the processor workloads.

The distribution information can be organized into an ordered list by per processor per entry. Each processor knows its own position in the list by its identity. Therefore each processor can act according to its own situation. When workloads are significantly different at a checkpoint, redistribution is possible and the balancing process begins. In the balancing process, the processor that has the biggest workload chooses the buckets whose size is closest to half of the difference between the biggest workload and the smallest workload, and ships these buckets to the processor which has the smallest workload. The processor transferring the buckets then changes the bucket addresses in the distribution information list. Since every other processor and data server has the same information and does the same calculation, they carry out the actions necessary for the balancing process without communication. All the data servers and other processors make the same changes to their bucket addresses simultaneously. When the data servers make the changes, the next partitioned data for the buckets will be sent to the new processor. The recalculation can take place based on the new distribution list in order to generate the new workload distribution.

4 Analysis

In order to compare the performance of the DBJ algorithm with that of other balancing algorithms, we develop a cost function with the same assumptions as those of Hua & Lee [3]. As there, we assume that the communication, disk I/O and CPU computation of each phase are perfectly overlapped, the two source relations have the same size and the same data distribution and are initially evenly distributed among disks, and the number of data servers equals the number of processors. Let N be $P + 1$ for the number of processing components. In the join operation, we further assume as do Hua & Lee [3] that under the skew condition

the relations are evenly distributed among all processors, except one which has excess data. It has $\sigma \times 100\%$ more tuples than each of the remaining processors. We have $|P_s| = \frac{|R|}{1+(N-1)(1-\sigma)}$ and $|P_u| = \frac{(1-\sigma)|R|}{1+(N-1)(1-\sigma)}$, where $|P_s|$ and $|P_u|$ are the cardinalities of the skewed partition and unskewed partitions respectively and $|R|$ is the cardinality of relation R . We assume that the overhead of the balancing process of the DBJ algorithm is 1% of the partition phase communication cost. Study of the implemented system in Section 5 suggests that this is an over estimate. The checkpoint β is set to 50% at the half way point of the partition phase.

The following additional parameters are used, following Hua & Lee [3]: t is the size in bytes of each tuple, ω_{io} is the I/O bandwidth of disk, ω_{comm} is the communication bandwidth, M is the memory capacity, I_{cpu} is the CPU path-length for processing a tuple, and μ is the CPU processing rate in instructions per second. The join time of the DBJ algorithm can then be computed as follows (each cost being time in seconds):

$$T_{dbj} = T_{part} + T_{join}$$

The total cost equals the sum of the costs of the partition phase and join phase.

$$T_{part} = \max(T_{pio}, T_{pcpu}, T_{pcomm})$$

The cost of the partition phase is dominated by the biggest cost of the communication, or CPU, or I/O operations.

$$T_{pio} = \frac{t|R|}{N\omega_{io}} + \frac{\frac{t|R|}{N} - M}{\omega_{io}}$$

The I/O cost in the partition phase is the sum of the costs of reading the first relation data from disk and writing data exceeding the memory capacity back to disk.

$$T_{pcpu} = |P_s| \times \frac{I_{cpu}}{\mu}$$

The processor with the skewed partition processes more data; processing this dominates the CPU cost in the partition phase.

$$T_{pcomm} = \left(\frac{N-1}{N} \frac{t|R|}{N\omega_{comm}} + T_{pextra} \right) \frac{101}{100}$$

The communication cost equals the cost of normal data distribution plus the cost of the balancing process plus the cost of the balancing process overhead.

$$T_{pextra} = \frac{(|P_s| - |P_u|)\beta t}{\omega_{comm}}$$

The extra cost of the balancing process equals the cost of the excess data moving to other processors. Since the balancing process happens before the partition phase finished, only a portion of the excess data is moved.

$$T_{join} = \lceil \frac{t|R|}{NM} \rceil (T_{jhash} + T_{jprobe})$$

The cost of the join phase is the sum of the cost of building the hash table and the cost of probing the hash table multiplied by the number of time that is done.

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

The cost of building the hash table is dominated by I/O cost or CPU cost for processing the data in fully utilized memory.

$$T_{jprobe} = T_{jlocal} + T_{jremote}$$

The cost of probing the hash table equals the cost of probing the local data and the cost of probing the remote data for the second relation.

$$T_{jlocal} = \max\left(\frac{M}{N\omega_{io}}, \frac{M}{tN} \frac{I_{cpu}}{\mu}\right)$$

N percent of the second relation data is local each time.

$$T_{jremote} = \max\left(\frac{M(N-1)}{N\omega_{io}}, \frac{M(N-1)}{tN} \frac{I_{cpu}}{\mu}, \frac{M(N-1)}{N\omega_{comm}}\right)$$

The rest of the second relation data is accessed via the network.

The cost functions of the GRACE, ABJ, and ABJ+ algorithms are given by Hua & Lee [3]. Using these cost functions we set parameters as follows: $N = 64$, $|R| = 1000000$, $T = 200$, $M = 2048000$, $\mu = 20000000$, $\omega_{io} = 4096000$, $\omega_{comm} = 4096000$, and $I_{cpu} = 1000$. The relative performance results are shown in Fig. 1 when σ varies from 0.1 to 0.9. We can see that under the cost model developed the DBJ algorithm gives better performance than either GRACE or ABJ algorithms, and better performance than ABJ+ for skew up to 85%, which covers most practical situations.

5 Implementation

It has been noted in Section 2 that the DBJ algorithm is architecture independent. It could be implemented on both shared nothing and shared memory architectures. In fact the algorithm has been fully implemented using the ADEPT machine [11]. This is a MIMD DBMS architecture, implemented at present on a T800 transputer-based Meiko Computing Surface.

Under the architecture, the transputers are divided into three groups: data servers, processors and the scheduler. The data servers, which are directly linked to disk drives, concentrate on caching and organizing source relation data. The processors do the actual query processing. The scheduler collects user queries, optimizes them and allocates them to a group of processors. The transputers are configured to be close together by using all available communication links, so data messages travel the minimum possible distance. A thirteen transputer architecture is used for illustration in this paper, consisting of three data servers, nine processors and one scheduler. The implementation of the DBJ algorithm has been in C and OCCAM 2: OCCAM 2 is used for implementing the communication functions and C is used for implementing the other functions. Remaining DBMS code comes from the previous work of the ADEPT project [11].

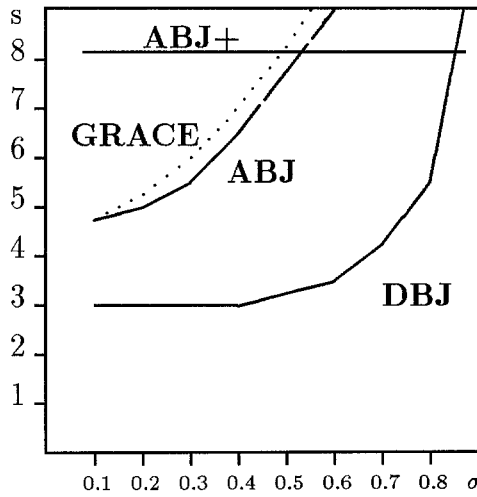


Fig. 1. Comparison

6 Experimental Results

In the following example, the relations are divided into 25 buckets and a join is carried out on 5 processors. The source relations are equally partitioned among three data servers and each tuple length is 70 bytes. Firstly, a description is given of how the real balancing process works in the algorithm. Suppose that relation R has 10200 tuples, with 3400 tuples being placed on each of three data servers; relation S has 20451 tuples and each data server has 6817 tuples, and a join operation joins the two relations on attributes $R.a$ and $S.b$. After the partition phase of the proposed hash join algorithm, the distribution of relation R without balancing is indicated in Table 1. Each column represents the distribution information for one processor. Suppose the first threshold value is at 70% of the partition phase and the average rate of growth of a bucket in the first 70% portion is maintained in the remaining 30% portion. The difference between the largest workload on processor 3 and the smallest workload on processor 1 is 720, and half of this is 360. This figure is bigger than 10% of the largest workload, and the gap will disappear if one of the smallest buckets is moved from processor 3 to processor 1. So the balancing process takes place by moving bucket 3 from processor 3 to processor 1 and the balanced distribution is showed in Table 1. Because the differences are now below the threshold value, no more balancing occurs.

Now compare the DBJ algorithm with an algorithm following the ABJ algorithm whereby the balancing process takes place statically after the partition phase, and which works as follows: each bucket of relation R is statically allocated to a processor in the partition phase, a designated coordinating processor collects the distribution information and reallocates the excess buckets to the

Initial Distribution						Balanced Distribution					
Proc No	1	2	3	4	5	Proc No	1	2	3	4	5
Load	1530	1800	2250	1800	1800	Load	1890	1800	1890	1800	1800
Buck No	1	2	3	4	5	Buck No	1	2	8	4	5
Tuples	360	360	360	360	360	Tuples	360	360	360	360	360
Buck No	6	7	8	9	10	Buck No	6	7	13	9	10
Tuples	360	360	360	360	360	Tuples	360	360	810	360	360
Buck No	11	12	13	14	15	Buck No	11	12	18	14	15
Tuples	360	360	810	360	360	Tuples	360	360	360	360	360
Buck No	16	17	18	19	20	Buck No	16	17	23	19	20
Tuples	90	360	360	360	360	Tuples	90	360	360	360	360
Buck No	21	22	23	24	25	Buck No	21	22		24	25
Tuples	360	360	360	360	360	Tuples	360	360		360	360
						Buck No	3				
						Tuples	360				

Table 1. Distribution Balancing

underflow processor after the partition phase is complete and then each processor performs the join as in the DBJ algorithm. Since the authors' system does not have hard disks, the disk drives on the data servers have been simulated. The comparative performance of the algorithms is shown in Fig. 2. The vertical axis is the join execution time and its unit is a second; the horizontal line represents the size of the source relation R . The solid line represents the DBJ algorithm's performance with threshold values of 70%, 80%, 90% and 100%, the dotted line represents the static algorithm's performance. Fig. 2 demonstrates that the DBJ algorithm's performance is better.

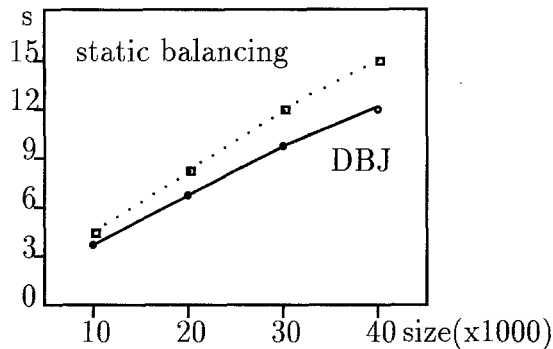


Fig. 2. Relation Size Influence

7 Conclusions

The *Dynamic Balancing Hash Join* (DBJ), has been proposed to handle the problem of skewed data in the join operation in multiprocessor database systems. The objective of this new algorithm is to avoid the high cost of preprocessing inherent in existing algorithms. The new algorithm only redistributes a small portion of the partitioned data and, thereby achieves a balanced output with little extra cost. This is achieved dynamically, without knowledge of the input distribution, nor any co-ordinating processor. A performance analysis shows that the new algorithm performs better than existing balancing hash join algorithms for a wide degree of skew.

References

1. David J. Dewitt, R. H. Gerber, "*Multiprocessor Hash Based Join Algorithms*", Proc. of the 11th VLDB Conference, Stockholm, Sweden, 1985.
2. David J. Dewitt, J.F. Naughton, D.A. Schneider, S. Seshadri, "*Practical Skew Handling in Parallel Joins*", Proc. of the 18th VLDB Conference, Vancouver, Canada, 1992.
3. Kien A. Hua, Chiang Lee, "*Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning*", Proc. of the 17th VLDB Conference, Barcelona, Spain, 1991.
4. Kitsuregawa, M., Tanaka, H., and T. Motooka, "*Application of Hash to Data Base Machine and its Architecture*", New Generation Computing, Vol:1, No. 1, 1983.
5. 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, Brisbane, Australia, 1990.
6. Hongjun Lu, Kian-Lee Tan, Ming-Chien Shan, "*Hash Based Join Algorithms for Multiprocessor Computers with Shared Memory*", Proc. of the 16th VLDB Conference, Brisbane, Australia, 1990.
7. Donovan A. Schneider and David J Dewitt, "*A Performance Evaluation of Four Parallel Join Algorithms in a Shared Nothing Multiprocessor Environment*", Proc. of the 1989 SIGMOD Conference, June, 1989.
8. C.B. Walton, A.G. Dale and R.M. Jenevein, "*A taxonomy and performance model of data skew effects in parallel joins*", Proc. of the 17th VLDB Conference, Barcelona, Spain, 1991.
9. G. D. Knott, "*Hashing function*", The Computer Journal, Vol. 18, No 3, 1973.
10. J.L. Wolf, D.M. Dias, P.S. Yu and J.J. Turek, "*An effective algorithm for parallelizing hash join in the presence of data skew*", IBM T.J. Watson Research Center Tech Report RC 15510, 1990.
11. R. G. Johnson, N. J. Martin, X. Zhao, "*The ADEPT Parallel Database Architecture and Implementation*", Proc. of the International Conference on Parallel Computing '91, London, September, 1991.
12. Witold Litwin, Marie-Anne Neimat, and Donovan A. Schneider, "*LH* — Linear Hashing for Distributed Files*", Proc. of the 1993 SIGMOD Conference, June, 1993.