Frequency-adaptive join for Shared Nothing machines

Mostafa Bamha and Gaétan Hains

LIFO, Université d'Orléans, B.P. 6759, 45067 Orléans Cedex 2, France
{bamha,ghains}@lifo.univ-orleans.fr

Abstract. Although many skew-handling algorithms have been proposed for simple join operations, they remain generally inefficient in the case of θ-join and in the case of multi-join. A new method for self-balancing equi-join operations on shared-nothing (SN) machines is proposed here. It offers deterministic and near-perfect load balancing through flexible control of communications in intra-transaction parallelism. The new algorithm mixes a balanced data-distribution strategy with pure hash-join. Its predictably low join-product- and attribute-value skews make it suitable for repeated use in multi-join operations. Its tradeoff between balancing overhead and speedup is analyzed in the BSP (*Bulk-synchronous parallel*) computing model. The scalable model predicts a negligible join product skew and a near-linear speed-up in any combination of selectivity, skew and number of processors. This prediction is confirmed by a series of tests.

Key words : PDBMS[1], Multi-join, Data skew, Join product skew, Dynamic load balancing.

1 Introduction

The appeal of parallel processing becomes very strong in applications which require ever higher performance and particularly in database applications such as : DBMS[2], data-warehousing, decision support and OLAP[3] [DMT98,BBH99].

The need for increased performance in database management systems is due to the size of data sets and the increasing complexity of queries [TB91]. Parallel processing is therefore a necessity and must be applied with internal and external data structures adapted to DBMS operations. Accordingly, there has been much work on the implementation of parallel DBMS (PDBMS): PRISMA/DB, Teradata, Gamma, Bubba and Parallel Oracle Server are examples of such systems. PDBMS can only maintain acceptable performance through efficient algorithms realizing complex queries on dynamic, irregular and distributed data. Such algorithms must dynamically reduce or eliminate data skew at the lowest possible cost because data skew can have a disastrous effect on performance [Yu90,KO90,HL91,HTY95,DNSS92,PI96,HK95,MMSY94,Mär98,BH98].

The work described here concentrates on the join operation, whose complexity is high enough to make it a prime target for parallelization. We apply an original balancing strategy whose efficiency is impossible to estimate without a realistic and portable cost model, whence our use of the BSP model.

This paper presents a new data-redistribution algorithm for equi-join operations in the presence of *attribute value skew*. It is adaptable to θ-join and efficient for multi-join. It computes exact frequency histograms to avoid *join product skew* i.e. to produce balanced outputs, thus minimizing subsequent redistribution time. A scalable and portable cost analysis is made with the BSP model, leading to general predictions about the effect of relation histograms on performance. The analysis suggests a hybrid *frequency-adaptive* algorithm, dynamically combining histogram-based balancing with standard hashing methods. The algorithm processes different sub-relations with one of two procedures according to their volume and hashing distribution, thus *adapting* to join attribute value *frequencies*. It's key feature is a data-distribution scheme whereby the intermediate distribution of tuples is balanced and whose planning of messages, a necessarily collective operation, is also balanced. The balancing operation thus uses the computing power of the whole network, including nodes where imbalance would be low.

[1] PDBMS : Parallel Database Management Systems.
[2] DBMS : Database Management Systems.
[3] OLAP : On Line Analytical Processing.

An earlier version of the algorithm has been presented in [BH98]. The present paper introduces a simplified and therefore more efficient algorithm, as well as more complete and realistic performance tests. The earlier version of the algorithm involved a parallel scan of frequency histograms (leading to so-called *cumulative histograms*) which has been found unnecessary. Also, the computation of so-called *sending orders*, a critical part of the balancing procedure, used a sorting algorithm which has been removed. Finally, the performance tests presented here use a fast sequential DBMS for high absolute performance.

The following section reviews necessary definitions and formalism. The next two describe our algorithm with its performance analysis and the experimental tests to confirm our predictions.

2 Parallel DBMS, join operations and the BSP model

In this section we state the parallel load balancing problem for join algorithms and the BSP execution-cost model used in later sections.

Load balancing in parallel join algorithms

The join $R \bowtie S$ of relations R, S[4] is generally costly because of the size of results which is generally large with respect to the relations' sizes. The size of the result of a join operation can be the product of the sizes of the input relations so that parallelization of this operation is highly desirable. Research has shown that the join is parallelizable with near-linear speed-up on scalable SN[5] architectures but only under ideal balancing conditions: *data skew* can have a disastrous effect on parallel performance [PI96]. Many algorithms have been proposed to handle data skew for a simple join operation, but little is known for the case of complex queries leading to multi-joins [LOT94,DNSS92,HL91]. In particular, the performance of PDBMS has generally been estimated on queries involving one or two join operations only [WFA95] thus ignoring the cumulative cost of data imbalance over the parallel machine's network. In fact, the problem of data skew is more acute with multi-joins because the imbalance of intermediate results is unknown during static query optimization. We address this question by designing a deterministic algorithm with near-perfect balancing properties and then estimating and measuring the overhead it incurs with respect to pure parallel hashing.

A number of skew-handling algorithms have been proposed to improve on the hash-join algorithms such as *Grace hash-join* [KTMo83]. Most of them defer the buckets allocation until the data partitioning is completed, the buckets are then assigned to processors based on their sizes. The *Bucket Spreading Parallel Hash Join* (BSJ) [KO90] is based on this approach. To ensure a balanced data load at each processor, BSJ uses an expensive specially designed network (Omega-Network) which does not scale well with the rest of the system [HL91,HLH95], The algorithm ignores JPS and moreover, the relations are redistributed twice. To improve on this algorithm, Hua et al. in [HL91,HLH95] introduced *Tuple interleaving parallel hash join* (TIJ). TIJ is similar to BSJ algorithm. The major difference is that instead of relying on a specially designed intelligent network for mapping buckets to processors, this decision is handled in software by a coordinator processor. Most of processors are idle during the tuning phase by the coordinator which limits the algorithm's scalability. The relations are also redistributed twice by TIJ and this algorithm also ignores JPS.

The authors of [PI96] have identified the two best proposed solutions in conventional- and sampling-based parallel join algorithms. Both are based on the notion of parallel hashing according to the value of the join attribute:

- In the first category of algorithms, the *extended adaptive load balancing parallel hash join* of [HL91] sends all tuples with the same attribute value to the same node. As a result, the algorithm may fail to balance the load for attribute-value distributions where a few values have a high

[4] R is usually called the *build* relation and S the *probe* relation

[5] SN : Shared Nothing multi-processor machines.

frequency. Moreover, the algorithm ignores the attribute value skew (AVS) of the *probe* relation and the join product skew (JPS). In other words, both the intermediate result of hashing (before local join computations) and the output relations (a dynamic and data-dependent subset of this intermediate result) can be unbalanced over the network, resulting in poor performance during and after a parallel join.

– In the second category of algorithms, *virtual process partitioning* [DNSS92] improves on the previous category of algorithms but fails to handle AVS in the probe relation once the build relation is chosen. It also ignores JPS.

To improve on these methods, the authors of [PI96] have then introduced an algorithm which minimizes the effect of the *expected* AVS. It precomputes histograms of the expected attribute-value distribution (i.e. statistically predicts the eventual result of hashing). However, it fails to correct JPS as follows. The method is based, like earlier ones, on hashing the set of attribute values into buckets, *each one being held by a single node.* The algorithm is therefore sensitive to the same effect as earlier methods: if a few attribute value account for most of the output relation, then most records of the *output* relation may end-up on the same node.

Another proposed method is a data redistribution algorithm for parallel join presented in [WTS98]. This algorithm only considers sets of join attribute values. It thus ignores value frequencies and therefore cannot handle high frequencies. Moreover, the algorithm is designed for hypercube computers: a less general model than BSP, and paper [WTS98] presents no experimental results to support the theoretical analysis.

We conclude that all existing methods are sensitive to imbalance when applied multiple times because of JPS. We propose a new solution to this problem: a deterministic method to avoid the effect of AVS and JPS at the cost of extra processing time. We analyze this overhead both theoretically and experimentally and conclude that it does not penalize overall performance because it is more than compensated by the gain in parallel speedup.

The BSP cost model

Bulk-Synchronous Parallel (BSP) computing is a parallel programming model introduced by Valiant [Val90] to offer a high degree of abstraction like PRAM models and yet allow portable and predictable performance on a wide variety of architectures. A BSP computer contains a set of processor-memory pairs, a communication network allowing inter-processor delivery of messages and a global synchronization unit which executes collective requests for a synchronization barrier. Its performance is characterized by 3 parameters expressed as multiples of the local processing speed: the number of processor-memory pairs p, the time l required for a global synchronization and the time g for collectively delivering a 1-relation (communication phase where every processor receives/sends at most one word). The network can deliver an h-relation in time $g * h$ for any arity h.

A BSP program is executed as a sequence of *supersteps*, each one divided into (at most) three successive and logically disjoint phases. In the first phase each processor uses its local data (only) to perform sequential computations and to request data transfers to/from other nodes. In the second phase the network delivers the requested data transfers and in the third phase a global synchronization barrier occurs, making the transferred data available for the next superstep (figure 1). The execution time of a superstep s is thus the sum of the maximal local processing time, of the global data delivery time and of the global synchronization time:

$$\text{Time}(s) = \max_{i:processor} w_i^{(s)} + \max_{i:processor} h_i^{(s)} * g + l$$

where $w_i^{(s)}$ = local processing time on processor i during superstep s and $h_i^{(s)} = \max\{h_{i+}^{(s)}, h_{i-}^{(s)}\}$ where $h_{i+}^{(s)}$ (resp. $h_{i-}^{(s)}$) is the number of words transmitted (resp. received) by processor i during superstep s. The execution time of a BSP program composed of S supersteps is therefore a sum of 3 terms:

$$\sum_{s \in [1..S]} \text{Time}(s) = W + H * g + S * l$$

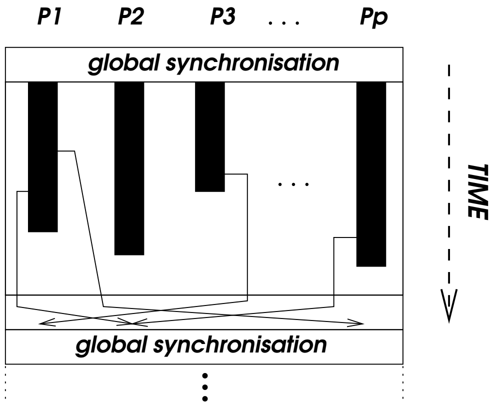


Fig. 1. A BSP superstep

where $W = \sum_s \max_i w_i^{(s)}$ and $H = \sum_s \max_i h_i^{(s)}$. In general W, H and S are functions of p and of the size of data n, or (as in the present application) of more complex parameters like data skew and histogram sizes. To minimize BSP execution time the algorithm design must jointly minimize the number S of supersteps, the sum H of communication concentrations and the length W of sequential work.

3 Parallel join operations and data skew

In PDBMS, relations are generally partitioned among processors by *horizontal fragmentation* with the values of a chosen attribute. Three methods are used [DG92]: hash partitioning, block-partitioning (also called range partitioning) and cyclic (also called round-robin) partitioning. Relation indices are also partitioned and partitioning is expected to be balanced. Fragmentation is physical for an SN machine and logical for a SD (*shared disk*) machine [RS95]. In the rest of the paper, R_i will denote the fragment of relation R placed on processor i.

The *θ-join* of two relations R and S on attribute A of R and attribute B of S (A and B of same domain) is the relation T, written $R \bowtie S$, obtained by concatenating the pairs of tuples from R and S for which $R.A \; \theta \; S.B$ where $\theta \in \{=, \neq, <, >, \leq, \geq\}$. When θ is $=$, the operation is called *equi-join*. The *semi-join* of S by R is the relation $S \ltimes R$ composed of the tuples of S which occur in the join of R and S. Semi-join reduces the size of S (when replacing it by $S \ltimes R$), a valid optimization because by definition $R \bowtie S = R \bowtie (S \ltimes R)$. We will write $\|R\|$ for the size of relation R in bytes and $|R|$ for its number of tuples.

Parallel equi-join usually proceeds in two phases: a redistribution phase by join attribute hashing and then sequential join of local fragments. Many such algorithms have been proposed. The principal ones are: *Sort-merge join*, *Simple-hash join*, *Grace-hash join* and *Hybrid-hash join* [SD89]. All of them (called hashing algorithms) are based on hashing functions which redistribute relations so that tuples having the same attribute value are forwarded to the same node. Local joins are then computed and their union is the output relation. Their major disadvantage is to be vulnerable to both *attribute value skew* (imbalance of the output of the first phase) and *join product skew* (imbalance of the output of local joins) [Yu90,PI96,LOT94]. The former affects immediate performance and the latter affects the efficiency of output (for example to distributed disks) or that of pipelined operations in the case of a multi-join.

To address this problem, we first introduce a new scheme for data distribution, useful for parallel equi-join of relations R when $\|Hist(R)\|$ is small, where $Hist(R)$ is the histogram of R, i.e. the list of pairs (d,n_d) over values d of the join attribute such that n_d is the number of tuples in R having this value. Entries for which $n_d = 0$ are not represented, whence the notion of *histogram size.* A small histogram indicates the presence of frequencies that are a large fraction of $\|R\|$ (Fig. 2, right). The exact threshold under which a histogram is considered to be small is determined by equation (2) in subsection 3.2.

Our scheme distributes data uniformly among processors and so avoids the effect of attribute value skew. Moreover it guarantees local join results of almost equal size and so avoids join product skew. We remark that join product skew can occur in the absence of attribute skew, thus highlighting the importance of avoiding both.

If, on the contrary, $\|Hist(R)\|$ is large then few frequencies should be large with respect to $\|R\|$ (Fig. 2, left). The majority of tuples, whose attribute values have low frequencies, then constitutes a relation with low data skew. Since equi-join by hashing yields quasi-linear speed-up for such a sub-relation [PI96], we can combine it with our data distribution scheme applied to the rest of the relation. The result is a general-purpose algorithm for equi-join, efficiently applicable to multi-join because it avoids join product skew.

By design, our algorithm is adaptable to general θ-join but we will not describe or analyze the general case here (except for an outline at the end of subsection 3.3) for the sake of clarity and because of the practical importance of equi-join.

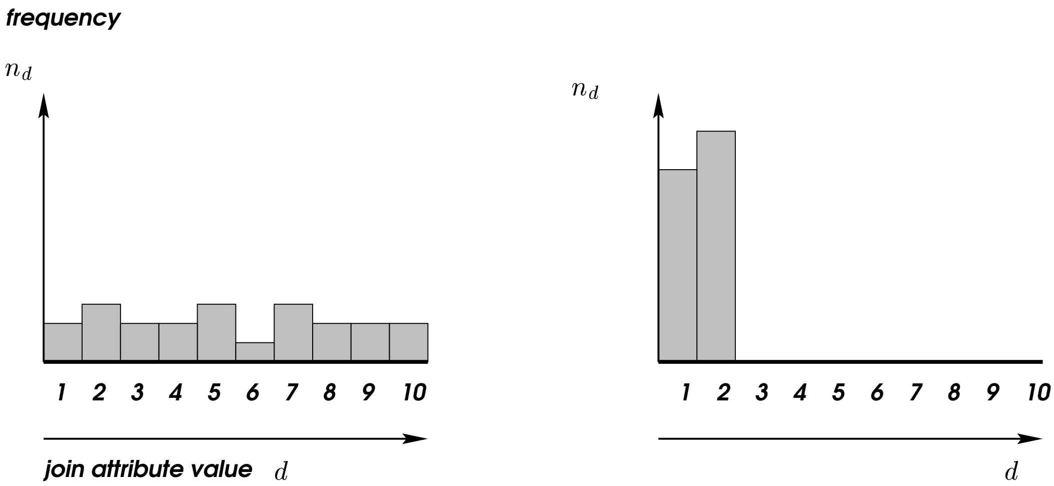


Fig. 2. Histograms for relation R count tuples by their value of the join attribute and dismiss those values for which the count is zero. As a result, a large histrogram (high $\|Hist(R)\|$, left) tends to occur when the relation is balanced. For the same relation size $\|R\|$ (histogram area), a small value of $\|Hist(R)\|$ (right) indicates the presence of some large frequencies and the need for balancing.

3.1 A new approach to data distribution

We first describe the redistribution method. It assumes a balanced data distribution for input relations, and leaves output relation fragments of almost the same size on every processor: $\|R_i\| \simeq \frac{\|R\|}{p}$ for $i \in [1..p]$. The following notation will be used:

R	is the relation to redistribute
R_i	is the sub-relation of R stored on processor i
$Hist(R)$	is the histogram of relation R (with respect to the join attribute value).
$Hist(R_i)$	is the histogram local sub-relation R_i (also stored on processor i).
$Hist_i(R)$	is processor i's part of the histogram of R
$Hist(R)(d)$	is the frequency (n_d) of value d in relation R.
$Hist(R_i)(d)$	is the frequency of value d in sub-relation R_i.

The computation of $Hist(R)$ will proceed in parallel and produce a balanced result so that $\|Hist_i(R)\| \simeq \|Hist(R)\|/p$. It is important to note that the network distribution of $Hist(R)$ (which determines $Hist_i(R)$) is completely independent of R's initial distribution (which determines R_i).

We will describe the algorithm while giving an upper bound on the BSP execution time of each phase. The $O(\ldots)$ notation only hides small constant factors: they depend on the implementation program but neither on data nor on the BSP machine parameters. Redistribution proceeds in 4 phases:

1. **Creating local histograms**, $Hist(R_i)_{i\in[1..p]}$, of blocks R_i by creating a local hash table of R_i. This phase costs

$$Time_{phase1} = O\left(\max_{i\in[1..p]} \|R_i\|\right).$$

Whenever the hash table of the fragments, R_i, does not fit into the local memory, in each processor i, we partition the fragment R_i into buckets using a common hash function in such a way that the generated buckets can fit into the available memory. These buckets are written back to disks. And the above process will be possible for each bucket.

2. **Creating the histogram of** R, $Hist(R)$. First by parallel hashing of the $Hist(R_i)$ in time $\max_{i\in[1..p]}(\|Hist(R_i)\|)$, the $Hist(R_i)$ are redistributed so that the complete histogram of R is evenly spread over the p processors. The cost of this is at most $O(g*\|Hist(R)\|+l)$. After hashing of destination addresses and communications are complete, each processor i merges the messages it received to construct $Hist_i(R)$ in time $O(\|Hist(R)\|)$. While merging, processor i also retains a trace of the network layout of the values d in its $Hist_i(R)$: this is nothing but the collection of messages it has just received. These data will be used in phase 3. In all, the creation and distribution of the global histogram is completed in

$$Time_{phase2} = O\left(\max_{i\in[1..p]} \|Hist(R_i)\| + g*\|Hist(R)\| + l\right)$$

with a balanced result: $\|Hist_i(R)\| \simeq \frac{\|Hist(R)\|}{p}$.

All the necessary information is then available and equally spread over the network. Redistribution is then performed by first computing routing information *with a balanced algorithm*. In this manner, the local time spent on preparing communications is a fixed part of the total work devoted to this task.

3. **Creation of communication templates :** This phase is performed jointly by all processors, *each one not necessarily in charge of computing its own messages* so as to balance the overall process. Processor i computes a set of necessary messages relating to the values d stored at i (values having non-zero entries in $Hist_i(R)$): the messages necessary to re-balance R's tuples having value d. Let $block_j(d)$ be the number of tuples of value d which processor j should store after redistribution of the R_i. Redistribution of R_i is performed to avoid JPS because the values which could lead to attribute value skew (those having high frequencies) are those which often cause the join product skew. It avoids thus, the slowdown usually caused by attribute value skew and the imbalance of the size of local joins processed by existing algorithms. Redistribution is done such that frequent values are evenly mapped to multiple nodes as follows.

if $\Big(Hist(R)(d) \bmod p = 0\Big)$ **then**

every processor j will receive an equal number $block_j(d) = \frac{Hist(R)(d)}{p}$ of tuples of value d.

else

begin

- Pick a random value j_0 between 0 and $(p-1)$
- **if** processor index j is between j_0 and $j_0 + (Hist(R)(d) \bmod p)$ **then**
  the processor of index j will receive a block of size : $block_j(d) = \lfloor \frac{Hist(R)(d)}{p} \rfloor + 1$
  else
  processor of index j will receive a block of size : $block_j(d) = \lfloor \frac{Hist(R)(d)}{p} \rfloor$.

end.

where $\lfloor x \rfloor$ is the largest integral value not greater than x.

The absolute value of $Rest_j(d) = Hist_j(d) - block_j(d)$ determines the number of tuples of value d which processor j must send (if $Rest_j(d) > 0$) or receive (if $Rest_j(d) < 0$).

For $d \in Hist_i(R)$ processor i holds in local memory a description of the layout of tuples of value d over the network (as noted in phase 2). It may therefore determine the number of tuples of value d which every processor must send/receive. Only those j for which $Rest_j(d) > 0$ (resp. < 0) send (resp. receive) tuples of value d. Phase 3 is thus completed in time $Time_{phase3} = O(\|Hist(R)\|)$.

4. **Data redistribution:** Every processor i holds, for every one of its local $d \in Hist_i(R)$, the non-zero communication volumes it prescribes as a part of communication template: $Rest_j(d) \neq 0$ for $j \in [1..p]$. This information will take the form of *sending orders* sent to their target in a first superstep, followed then by the actual redistribution superstep where processors obey all orders they have received.
Each processor i first splits the processors indices j in two groups : those for which $Rest_j(d) > 0$ and those for which $Rest_j(d) < 0$. This is done by a sequential traversal of the $Rest_{..}(d)$ array.

Let α (resp. β) be the number of j's where $Rest_j(d)$ is positive (resp. negative) and $Proc(k)_{k\in[1..(\alpha+\beta)]}$ the array of processor indices for which $Rest_j(d) \neq 0$ in the manner that :

$$\begin{cases} Rest_{Proc(j)}(d) > 0 & for \quad j \in [1..\alpha] \\ Rest_{Proc(j)}(d) < 0 & for \quad j \in [(\alpha+1)..\beta]. \end{cases}$$

A sequential traversal of $Proc(k)_{k\in[1..(\alpha+\beta)]}$ determines the number of tuples each processor j will send. The sending orders concerning attribute value d are computed by the following procedure:

$i := 1;\quad j := \alpha + 1;$
while $(i \leq \alpha)$ `do`
begin
* n_tuples=min$(Rest_{Proc(i)}(d), -Rest_{Proc(j)}(d))$;
* `order_to_send`(Proc(i),Proc(j),d,n_tuples);
* $Rest_{Proc(i)}(d) := Rest_{Proc(i)}(d)-$ n_tuples;
* $Rest_{Proc(j)}(d) := Rest_{Proc(j)}(d)+$ n_tuples;
* `if` $Rest_{Proc(i)}(d) = 0$ `then` $i := i + 1$; `fi`
* `if` $Rest_{Proc(j)}(d) = 0$ `then` $j := j + 1$; `fi`
end.

of complexity $O(\|Hist(R)\|)$. For a given d, less than $(p-1)$ processors can send data. For every i and $d \in Hist_i(R)$ the orders `order_to_send`$(j, ...)$ are sent to processor j when $j \neq i$ in time

$O\left(g * \|Hist(R)\| + l\right)$. Every processor j is then informed of the required redistribution of its tuples R_j, which it then performs at cost $O(g * \|R_i\| + l)$. In all, phase 4 costs:

$$Time_{phase4} = O\left(g * (\max_{i \in [1..p]} \|R_i\| + \|Hist(R)\|) + \|Hist(R)\| + l\right).$$

and the complete redistribution algorithm:

$$Time_{redistribution} = O\left((1+g) * (\max_{i \in [1..p]} \|R_i\| + \|Hist(R)\|) + l\right).$$

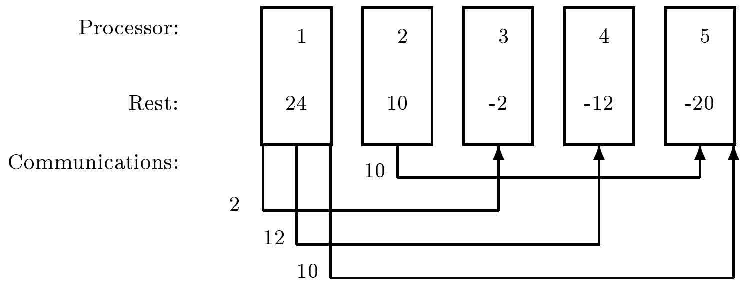


Fig. 3. Sending orders as a function of $Rest$ values.

Figure 3 gives an example of $Rest$ values and corresponding sending orders.

3.2 Join processing ("balancing" algorithm)

To compute the join of relations R, S we first select among them the inner- or *build relation* used to create the hashing table, and the outer- or *probe relation* used to access the hash table. To minimize the cost of redistribution the build relation $T \in \{R, S\}$ is chosen so that $\frac{\|Hist(T)\|}{\|T\|}$ is least. This selection costs:

$$Time_a = O\left(\max_{i \in [1..p]} (\|R_i\| + \|S_i\|) + (l+g) * \log(p)\right).$$

Assume $T = R$. Redistribution of R is then made with cost $Time_{redistribution}$ as estimated above. Local joins can then be computed. Taking advantage of the identities:

$$\begin{aligned} R \bowtie S &= \bigcup_{i,j} R_i \bowtie S_j = \bigcup_{i,j} R_i \bowtie (S_j \ltimes Hist(R)) \\ &= \bigcup_i R_i \bowtie \bigcup_j (S_j \ltimes Hist(R)) = \bigcup_i R_i \bowtie (S \ltimes Hist(R)), \end{aligned}$$

we first compute local semi-joins of S_j with $Hist(R)$ in time: $Time_b = O\left(\max_{j \in [1..p]} \|S_j\|\right)$ and then broadcast the semi-join results (generally very small with respect to the local relation size) in time:

$$Time_c = O\Big(\sum_{j=1}^{j=p} g * \|S_j \ltimes Hist(R)\| + l\Big) = O\Big(g * \|S \ltimes Hist(R)\| + l\Big).$$

Finally, local joins are computed in time:

$$Time_d = O\left(\max_{i \in [1..p]} \|R_i\| + \|S \ltimes Hist(R)\| + \frac{1}{p} * \|R \bowtie S\|\right).$$

In total join has therefore taken the sum of the 4 costs above with that of redistribution:

$$Time_{join} = O\Big((1+g)*(\max_{i\in[1..p]}\|R_i\| + \|Hist(R)\|) + \max_{i\in[1..p]}\|S_i\| \\ +(1+g)*\|S \rhd\!\lhd Hist(R)\| + \frac{1}{p}*\|R \bowtie S\| + (g+l)*\log(p)\Big). \quad (1)$$

The local join results are all of balanced size: approximately $\frac{\|R \bowtie S\|}{p}$. Observe that sequential join processing would require at least the time for input and output: $bound_{inf_1} = \Omega\left(\|R\| + \|S\| + \|R \bowtie S\|\right)$. Parallel processing with p processors requires therefore $bound_{inf_p} = \frac{1}{p} * \Omega\left(\|R\| + \|S\| + \|R \bowtie S\|\right)$ and the balancing algorithm has optimal asymptotic complexity when

$$\|Hist(R)\| \leq \max(\frac{\|R\|}{p}, \frac{\|S\|}{p}, \frac{\|R \bowtie S\|}{p}) \quad (2)$$

This is due to the fact that its results of local joins are of size $\simeq \frac{\|R \bowtie S\|}{p}$ and the cost of redistribution is bounded above by terms in $bound_{inf_p}$.

Remark : For large histograms $\|Hist(R)\| \simeq \|R\|$, this algorithm is inefficient but it is also not always necessary: hashing algorithms (*Simple-hash join*, *Grace-hash join* and *Hybrid-hash join* [SD89]) can then provide near-linear speed-up, if skew is small: so we will combine them with the balancing algorithm.

3.3 Join operations in the presence of attribute value skew ("frequency-adaptive")

This hybrid algorithm proceeds in two steps. The first one temporarily removes tuples whose values contribute to attribute value skew, those occurring more often than a threshold frequency (for example $k * p * \log p$) and then performs the join of the remaining sub-relations R', S' with known parallel hashing algorithms [SD89]. The second step performs another join, using only $R - R'$ and $S - S'$ i.e. those tuples contributing to the imbalance, using the balancing algorithm (which will be efficient in the presence of high frequencies, cf. inequation (2)). This method avoids the slowdown usually caused by attribute value skew and balances the size of local joins processed by existing algorithms. The join is then the union of the results of both steps.

This frequency-adaptive algorithm avoids join product skew because the values which could lead to attribute value skew (those having high frequencies) are those which often cause join product skew in hashing algorithms. It presents quasi-linear speed-up because the first step is known to have this property [PI96] and because the second step is also very efficient (cf. section 3.2).

A possible improvement to the frequency-adaptive algorithm would be to compute the parameter k above in a dynamic fashion based on the machine parameters, processor loads, `build_time` when a tuple was inserted in the hash table, `probe_time` when a tuple was probed for in the hash table, etc. The tests presented below have used a fixed value of $k = 1$.

Finally, we outline the adaptation of our algorithm to θ-join operations, where $\theta \in \{\neq, <, >, \leq, \geq\}$. In this case, we only split the relation R into two sub-relations R' and R'' in the manner that $R = R' \cup R''$ where the tuples in relation R' (resp. R'') are associated to low (resp. high) frequencies for the join attribute values. The θ-join of R and S will be thus: $R \bowtie_\theta S = (R' \bowtie_\theta S) \cup (R'' \bowtie_\theta S)$. The first θ-join is computed using one of the hashing algorithms while the second is computed using the *balancing-algorithm*.

4 Experimental results

The frequency-adaptive algorithm (fa_join) for equi-join has been compared for speed-up and skew with pure hash-join (the "standard method" or std_join). Tests have used a Fujitsu AP-3000 shared-nothing architecture (based on Sparc Ultra-1 processors with 512Mb of memory) using MySQL-3.22[6] database

[6] MySQL home page "http://www.tcx.se".

servers for local relational processing and MPI-1.1[7] as a Message Passing Interface for communications. Although the number of processors was relatively small — a maximum of 12 were available — all theoretical and experimental evidence indicates that the gain for using fa_join *increases* with machine size beyond this limit.

Frequencies of join attribute values have been taken to follow the Zipf distribution [Zip49,Chr84,LT92] as it is the case in most database tests. Given this distribution, in a relation R with a domain $\{1, 2, \ldots, D\}$ of distinct values for its join attribute, the i^{th} distinct value has a number of tuples given by the following expression :

$$|v_i| = \frac{|R|}{i^z * \sum_{j=1}^{D} \frac{1}{j^z}}$$

where z is the skew factor. When $z = 0$ the distribution is uniform and when $z \geq 1$ it becomes highly skewed.

We studied the effect of the correlation between the skew values in the two relations. Two types of correlation are modeled: *Ordered correlation* and *random correlation*. For ordered correlation, the join attribute values in both relations have the same ranking sequences: the highest ranked value in attribute R_A of relation R is also the highest ranked value in the corresponding attribute S_A in relation S. A more general situation is modeled by "random correlation": the attribute values in R and S are randomly correlated. To model the two correlation types, we introduced a correlation factor, called "corr", between R and S. A value 0 of *corr* means that relations R and S are randomly correlated and a value i of *corr* means that the i highest ranked values are in ordered correlation and the remaining values are randomly correlated. Hence increasing the value of *corr* increases the join selectivity[8]

We have performed all tests on relations containing $100K$ tuples of 100 bytes each, attribute value skew has followed variations of the Zipf distribution's z parameter from 0 to 2 and the correlation factor has been varied from 0 to 100. The join selectivity has measured to vary from 0.00% to 0.15% as a result of variations in the correlation factor.

We have measured the JPS by the maximum deviation of the local join result size with respect to the average over all processors. For simplicity, all figures and observations below will refer to this value as "JPS". For example, a JPS value of 100% will mean that some local sub-relations of the join result are twice as large as the average over all processors.

4.1 Speed-up test

The speed-up test confirms the absence of a net overhead for using frequency-adaptive join. It has been performed on relations with a slight attribute value skew (Zipf distribution's z parameter = 0.5) and of total join size $\simeq 3M$ tuples. The number of processors has varied from 1 to 12. Timings show an almost linear speed-up for the fa_join algorithm which is faster than std_join (Fig. 4.a) *even for this low value of* z. Moreover the join product skew of std_join is relatively large with respect to relation size, while it is negligible for the fa_join algorithm. This is shown by Fig. 4.b and also illustrates the fact that an inefficient algorithm can generate a large JPS even for low attribute value skew. As a result, when processing multiple join operations the cost of redistribution will be much smaller with the fa_join algorithm than with the std_join algorithm.

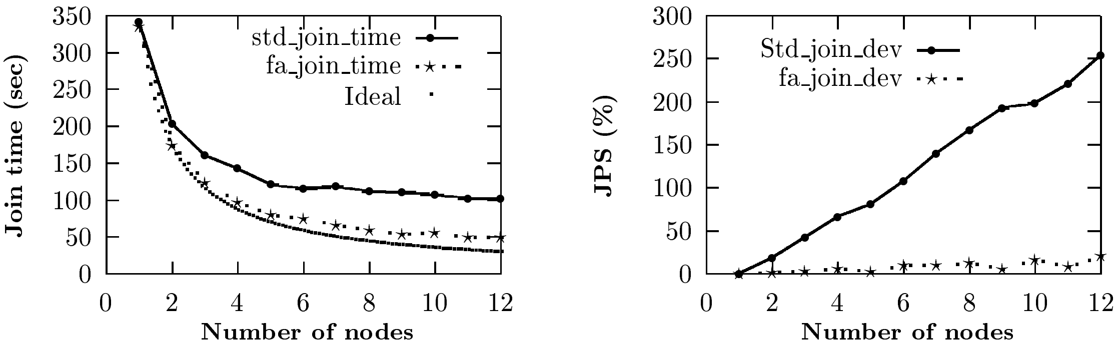
4.2 Attribute value skew test

Two other tests were performed to study the effect of attribute value skew on performance. The number of processors has been fixed to 12, relations R, S have been given sizes of $100K$ tuples.

The first of those tests used a fixed AVS of 0.5 for R while varying it in S from 0 to 2, yielding output relations of sizes between $700K$ and $15M$ tuples (i.e. selectivity between 0.007% and 0.15%). Timings show that the fa_join algorithm is vastly superior to the std_join algorithm while guaranteeing

[7] MPI-1.1 is available at "http://www-unix.mcs.anl.gov/mpi/".

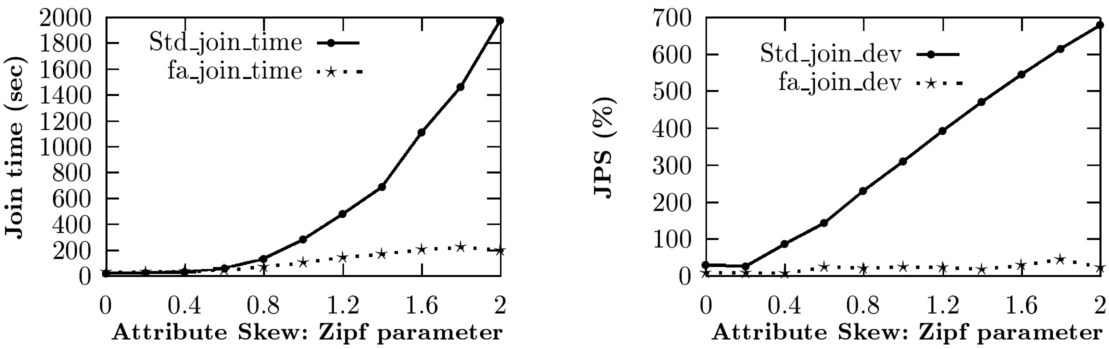
[8] selectivity=$\frac{|R \bowtie S|}{|R| * |S|}$ where $|T|$ denotes the number of tuples of relation T.



(a) (b)

Fig. 4. Speed_up performance for small attribute skew.

a balanced output, which confirms its suitability in the presence of any amount of skew (Fig. 5).



(a) (b)

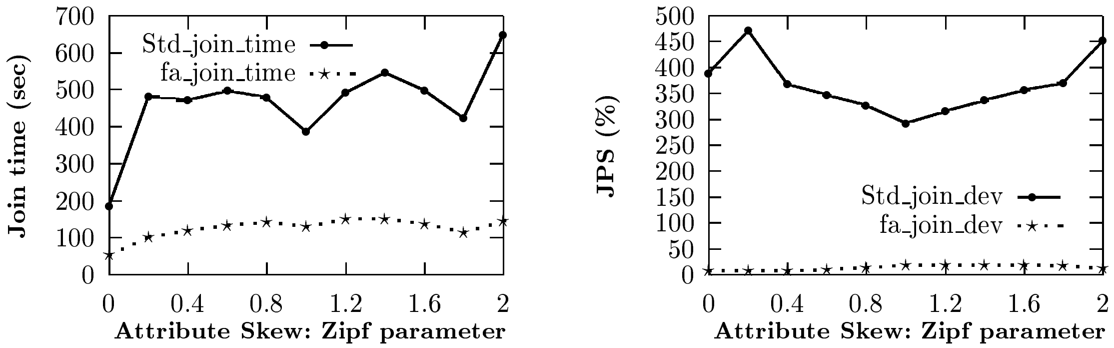
Fig. 5. Effect of attribute value skew for the probe relation.

The second test used a highly skewed relation R with a fixed skew of 2 while varying skew factor in S from 0 to 2, yielding output relations of sizes between $3.5M$ and $12M$ tuples. Figure 6.b shows that the fa_join algorithm almost completely eliminates JPS while std_join produces relatively high JPS: higher than 300%. This means that certain local sub-relations produced by std_join are 300% above the average size, i.e. four times larger than in a balanced situation. Moreover, the large size of output relations relative to the input means that the local processing of output relation fragments dominates processing time for this test. For fa_join, this time is given by term $\frac{1}{p}\|R \bowtie S\|$ in equation (1). For std_join, the imbalanced output relation multiplies this time by a factor proportional to (JPS+100%) as explained above. As a result, processing time for std_join is approximately four times as high as processing time for fa_join in Fig 6.a.

We also observe that fa_join speed is almost independent of the attribute skew. Because of random correlation (explained above), our tests generate irregular join product sizes whenever AVS varies. As a result, the speed of fa_join shown on figure 6.a appears to increase slightly with AVS. But the variation can be attributed to join product size variations.

4.3 Join selectivity test

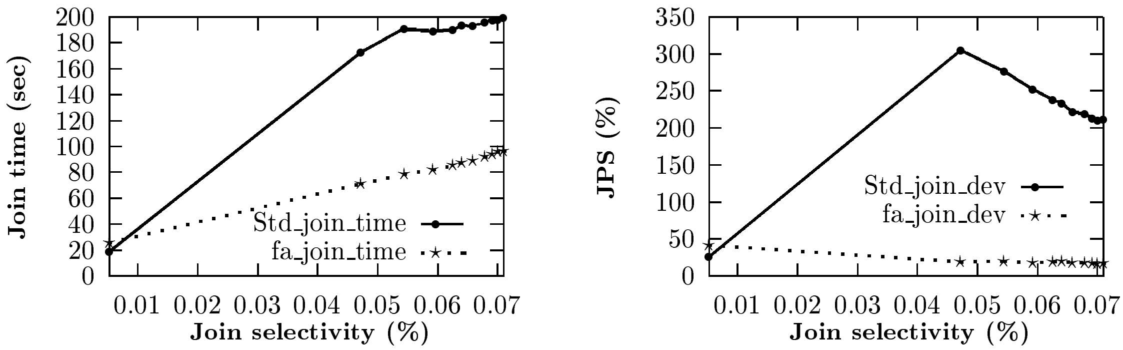
This last test measured the effect of join selectivity on performance in the same condition as the speed-up test (very small attribute value skew). The number of processors has been fixed to 12, relations



(a) (b)

Fig. 6. Effect of high skew on both build and probe relations.

R, S were the same as in the speed-up test, the join selectivity has been made to vary from 0.005% to 0.07% (through the correlation factor), yielding output relations of sizes between $500K$ and $7M$ tuples. Results show that fa_join is faster than std_join (Fig. 7.a) at every value of join selectivity, while guaranteeing a very low and almost constant *JPS* (Fig. 7.b). This confirms and strengthens the results of the speed-up test.



(a) (b)

Fig. 7. Effect of join selectivity.

4.4 Comments

The above tests confirm our theoretical predictions about processing overhead, attribute value skew and join product skew. They strengthen our earlier results [BH98] which were conducted with relatively small relations and using a very slow sequential program (Mini-SQL).

5 Conclusions

We have presented a parallel redistribution algorithm fa_join, for equi-join processing on SN machines, efficiently compensating attribute value skew while guaranteeing the near-absence of join product skew. It reduces the cost of redistribution for intermediate results in multi-joins and thus improves their performance which has been vulnerable to data imbalance. To our knowledge it is the first parallel algorithm whose efficiency is preserved for multi-join. Our experiments confirm that in all cases the fa_join algorithm has created a minimal imbalance, at no extra cost. The method used here is also applicable to shared-disk machines but speed-up will then be limited by the I/O bottleneck.

In their survey of skew handling algorithms, Lu, Ooi and Tan (conclusion of [LOT94]) mentioned the

> [...] urgent need for algorithms that can balance the join load dynamically, this is, during join computation. It may also be beneficial to dynamically rearrange an intermediate relation if the overhead of reorganization is less than its benefit.

The BSP cost analysis and implementation presented here are conclusive proofs of a general and portable solution to this overhead-benefit tradeoff. The algorithm they suggest offers data-independent scalability. We expect to show in the future that the algorithm is adaptable to θ-join when $\theta \neq =$, as outlined at the end of section 3.3

Acknowledgement

We thank FECIT[9] for free access to an AP-3000 system.

References

[BBH99] M. Bamha, F. Bentayeb, and G. Hains. An efficient scalable parallel view maintenance algorithm for shared nothing multi-processor machines. In *the 10th International Conference on Database and Expert Systems Applications DEXA'99*, volume 1677 of *Lecture Notes in Computer Science*, pages 616–625, Florence, Italy, 1999. Springer-Verlag.

[BH98] M. Bamha and G. Hains. A self-balancing join algorithm for SN machines. In Y. Pan, S. G. Akl, and K. Li, editors, *Proceedings of International Conference on Parallel and Distributed Computing and Systems (PDCS)*, pages 285–290, Las Vegas, Nevada, USA, October 1998. IASTED/ACTA Press.

[Chr84] S. Christodoulakis. Implications of certain assumptions in database performance evaluation. *ACM Transactions on Database Systems*, 9(2):163–186, 1984.

[DG92] D. J. Dewitt and J. Gray. Parallel database systems : The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[DMT98] A. Datta, B. Moon, and H. Thomas. A case for parallelism in datawarehousing and OLAP. In *Ninth International Workshop on Database and Expert Systems Applications, DEXA 98*, IEEE Computer Society, pages 226–231, Vienna, 1998.

[DNSS92] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical Skew Handling in Parallel Joins. In *Proceedings of the 18th VLDB Conference*, pages 27–40, Vancouver, British Columbia, Canada, 1992.

[HK95] L. Harada and M. Kitsuregawa. Dynamic join product skew handling for hash-joins in shared-nothing database systems. In *Fourth International Conference on Database Systems for Advanced Applications*, pages 246–255, 1995.

[HL91] K. A. Hua and C. Lee. Handling data skew in multiprocessor database computers using partition tuning. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *Proc. of the 17th International Conference on Very Large Data Bases*, pages 525–535, Barcelona, Catalonia, Spain, 1991. Morgan Kaufmann.

[HLH95] Kien A. Hua, Chiang Lee, and Chau M. Hua. Dynamic load balancing in multicomputer database systems using partition tuning. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):968–983, 1995.

[HTY95] K. A. Hua, W. Tavanapong, and H. C. Young. A Performance Evaluation of Load Balancing Techniques for Join Operations on Multicomputer Database systems. In P. S. Yu and A. L. P. Chen, editors, *Proceedings of the 11th International Conference on Data Engineering*, pages 44–51, Los Alamos, CA, USA, 1995. IEEE Computer Society Press.

[KO90] M. Kitsuregawa and Y. Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for skew in the super database computer (SDC). *Very Large Data Bases: 16th International Conference on Very Large Data Bases, August 13–16, Brisbane, Australia*, pages 210–221, 1990.

[KTMo83] M. Kitsuregawa, H. Tanaka, and T. Moto-oka. Application of hash to database machine and its architecture. *New Generation Computing*, 1(1):63–74, 1983.

[9] FECIT : Fujitsu European Center for Information and Technology.

[LOT94] H. Lu, B.-C. Ooi, and K.-L. Tan. *Query Processing in Parallel Relational Database Systems.* IEEE Computer Society Press, Los Alamos, California, 1994.

[LT92] H. Lu and K.-L. Tan. Dynamic and load-balanced task-oriented database query processing in parallel systems. In *Proceedings of Advances in Database Technology (EDBT '92)*, volume 580 of *LNCS*, pages 357–372, Berlin, Germany, 1992. Springer.

[Mär98] H. Märtens. Skew-insensitive join processing in shared-disk database systems. *Proc. of Issues and Applications of Database Technology (IADT '98), Berlin*, pages 17–24, July 1998.

[MMSY94] A. N. Mourad, R. J. T. Morris, A. Swami, and H. C. Young. Limits of parallelism in hash join algorithms. *Performance evaluation*, 20(1/3):301–316, May 1994.

[PI96] V. Poosala and Y. E. Ioannidis. Estimation of query-result distribution and its application in parallel-join load balancing. *In: Proc. 22th Int. Conference on Very Large Database Systems, VLDB'96, pp. 448-459, Bombay, India*, September 1996.

[RS95] E. Rahm and T. Stöhr. Analysis of parallel scan processing in shared disk database systems. *Proc. of EURO-PAR'95 Conference, Stockholm, Springer-Verlag, LNCS*, pages 485–500, August 1995.

[SD89] D. Schneider and D. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In J. Clifford, B. Lindsay, and D. Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data, Portland, Oregon*, pages 110–121, New York, NY 10036, USA, 1989. ACM Press.

[TB91] W. B. Teeuv and H. M. Blanken. Complex object join in a distributed database. Technical report, University of Twente, Departement of Computer Science, The Netherlands, 1991.

[Val90] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[WFA95] A. N. Wilschut, J. Flokstra, and P. M.G. Apers. Parallel evaluation of multi-join queries. *Procedings of the ACM-SIGMOD*, 24(2):115–126, 1995.

[WTS98] R. Wong, R. Topor, and H. Shen. A parallel sort-balance mutual range-join algorithm on hypercube computers. *Microprocessors And Microsystems*, 22(3-4):209–215, 1998.

[Yu90] M. Seethaand P. S. Yu. Effectiveness of parallel joins. *IEEE, Trans. Knowledge and Data Enginneerings*, 2(4):410–424, December 1990.

[Zip49] G. K. Zipf. Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology. *Reading, MA, Adisson-Wesley*, 1949.