A skew-insensitive algorithm for join and multi-join operations on Shared Nothing machines

M. Bamha and G. Hains
{bamha,ghains}@lifo.univ-orleans.fr

LIFO, Université d'Orléans, B.P. 6759, 45067 Orléans Cedex 2, France.

Abstract. Join is an expensive and frequently used operation whose parallelization is highly desirable. However effectiveness of parallel joins depends on the ability to evenly divide load among processors. Data skew can have a disastrous effect on performance. Although many skew-handling algorithms have been proposed they remain generally inefficient in the case of multi-joins due to join product skew, costly and unnecessary redistribution and communication costs. A parallel join algorithm called fa_join has been introduced in an earlier paper with deterministic and near-perfect balancing properties. Despite its advantages, fa_join is sensitive to the correlation of the attribute value distributions in both relations. We present here an improved version of the algorithm called Sfa_join with a symmetric treatment of both relations. Its predictably low join-product and attribute-value skew makes it suitable for repeated use in multi-join operations. Its performance is analyzed theoretically and experimentally, to confirm its linear speed-up and its superiority over fa_join.

1 Introduction

Research has shown that the join operation is parallelizable with near-linear speed-up on shared nothing machines only under ideal balancing conditions. Data skew can have a disastrous effect on performance [1, 3, 13, 9, 4, 11, 5, 10].

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 [8, 4, 7, 11]. In particular, the performance of PDBMS has generally been estimated on queries involving one or two join operations only [15]. However the problem of data skew is more acute with multi-joins because the imbalance of intermediate results is unknown during static query optimization [8]. Join algorithms described in previous works induce costly and unnecessary communications. In this paper we show that the communications involved in computing the join $R \bowtie S$ of two relations R and S can be reduced to a minimum by :

- an efficient computation of the semi-joins $R \ltimes S$ and $S \ltimes R$. This reduces the redistribution cost to only the tuples of relations R and S which will effectively be present in the join result.
- a dynamic choice of the build[1] and probe[2] relation for each bucket (sub-relation) of the two relations allowing to reduce the sub-set replication costs while avoiding attribute and join product skews.

[1] Build relation : the relation used to create the hash table.
[2] Probe relation : the relation used to probe the hash table.

Our present contributions to parallel relational query processing are:
1. A new parallel join algorithm called *sfa_join* (symmetric frequency adaptive join algorithm) for shared nothing machines. This algorithm has near-perfect balancing properties and supports flexible control of communications induced by intra-transaction parallelism. The sfa_join algorithm is based on:
(a) a symmetric partial duplication of data. (b) an improved version of the redistribution algorithm of *fa-join* [3, 1] which efficiently avoids the problem of attribute value- and join product skews.
2. An analysis of sfa_join in the scalable and portable BSP cost model. It predicts a negligible join product skew and a near linear speed-up for our algorithm, independently of the data and of the (shared nothing) architecture's bandwidth, latency and number of processors. This prediction is confirmed by a series of tests. Some of the tests also confirm the superiority of sfa_join over fa_join. Its suitability for multi-joins follows from the output's near-perfect balancing *at no extra processing cost.*

2 Load balancing in parallel joins and the BSP model

Parallel 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* [12]. 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) [13, 11, 8]. The former affects immediate performance and the latter affects the efficiency of output or pipelined operations in the case of a multi-join.

The authors of [11] have identified the two best proposed solutions in conventional and sampling-based parallel join algorithms. Their study has allowed us to conclude [2] that all existing methods are sensitive to imbalance when applied multiple times because of JPS.
To address this problem, we introduced in [3] a deterministic data-redistribution algorithm with near-perfect balancing properties. It is adaptable to θ-join and efficient for multi-join. It dynamically computes *exact* frequency histograms to avoid *join product skew* (JPS). A scalable and portable cost analysis was 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 (*fa_join* algorithm) [3], dynamically combining histogram-based balancing with standard hashing methods. The *fa_join* algorithm avoids the slow-down usually caused by attribute value skew (AVS) and the imbalance of the size of local joins processed by the standard algorithms. It avoids AVS and JPS at the cost of extra processing. We analyzed this overhead both theoretically and experimentally and concluded that it does not penalize overall performance, even in the absence of skew.

The *fa_join* algorithm was thus designed to efficiently avoid the problem of AVS and JPS. However, its performance is less than ideal when computing the join of highly skewed relations because of unnecessary redistribution and communication costs. We introduce here a new parallel algorithm called *sfa_join* (Symmetric frequency adaptive join algorithm) to perform such joins. Its predictably low join-product and attribute-value skew make it suitable for repeated use in multi-join operations. Its performance is analyzed using the BSP cost model which predicts for it a near-linear speedup.

Bulk-Synchronous Parallelism (BSP) is a parallel programming model introduced by Valiant [14] to offer a high degree of abstraction like PRAM models and yet allow portable and predictable performance. The performance of a parallel architecture 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 is assumed to deliver an h-relation in time $g * h$ for any arity h. The execution time of a parallel program is then the sum of 3 terms: W, the time spent by the most loaded processors on local computation between communication phases; $H = \sum_i g * h_i$ where h_i measures the i^{th} communication phase and $S * l$ where S is the number of such phases.

3 Data redistribution : A new approach

We first assume that relation R (resp. S) is partitioned among processors by horizontal fragmentation and the fragments R_i for $i = 1, .., p$ are almost of the same size on every processor, i.e. $|R_i| \simeq \frac{|R|}{p}$ where p is the number of processors. In the rest of this paper we use the following terminology for $T \in \{R, S\}$:

T_i denotes the fragment of relation T placed on processor i, $Hist(T)$ denotes the histogram of relation T with respect to the join attribute value, i.e. a list of pairs (v, n_v) where $n_v \neq 0$ is the number of tuples having the value v for the join attribute. The histogram is often much smaller and never larger than the relation it describes, $Hist(T_i)$ denotes the histogram of fragment T_i $Hist_i(T)$ is processor i's fragment of the histogram of T, $Hist(T)(d)$ is the frequency (n_d) of value d in relation T, $Hist(T_i)(d)$ is the frequency of value d in sub-relation T_i, and $|T|$ denotes the size of relation T.

We will outline the algorithm while giving an upper bound on the BSP execution time of each phase. A complete description can be found in [2]. Our redistribution algorithm is the basis for efficient and scalable processing. It proceeds in 5 phases:

Phase 1 : Creating local histograms
Local histograms $Hist(R_i)_{i=1,..,p}$ (resp. $Hist(S_i)_{i=1,..,p}$) of blocks R_i (resp. S_i) are created in parallel by a traversal of R_i (resp. S_i). In principle, this phase costs: $Time_{phase1} = O\left(\max_{i=1,..,p}\left(|R_i| + |S_i|\right)\right)$,
but in practice, the extra cost for this operation is negligible because the histograms can be computed on the fly while creating local hash tables.

Phase 2 : Creating the histogram of R $\bowtie$ S

The first step is to create the histograms $Hist(R)$ and $Hist(S)$ by a parallel hashing of the histograms $Hist(R_i)$ and $Hist(S_i)$. The histograms $Hist(R_i)$ and $Hist(S_i)$ are first redistributed so that the complete histograms of R and S are evenly spread over the p processors. The cost of redistribution is : $Time_{phase2.a} = O(g * (|Hist(R)| + |Hist(S)|) + l)$. After hashing of destination addresses and communications are complete, each processor i merges the messages it received to constitute $Hist_i(R)$ (and $Hist_i(S)$) in time :
$Time_{phase2.b} = O(|Hist(R)| + |Hist(S)|)$.

While merging, processor i also retains a trace of the network layout of the values d in its $Hist_i(R)$ (resp. $Hist_i(S)$): this is nothing but the collection of messages it has just received. These data will be used in phase 4.
The fragment $Hist_i(R \bowtie S)$ is then computed on each processor i, by "intersecting" $Hist_i(R)$ and $Hist_i(S)$, in time :
$Time_{phase2.c} = O\big(\max_{i=1,..,p}(min(|Hist_i(R)|, |Hist_i(S)|))\big)$.
While creating the fragments of $Hist(R \bowtie S)$ we store, for each value $d \in Hist_i(R \bowtie S)$, an extra information called *index* (*index(d)* $\in \{0, 1, 2\}$). This information will permit us to decide if, for a given value d, the frequencies of tuples of relations R and S having the value d are greater (resp. lesser) than a threshold frequency f_0. It also permit us to choose dynamically the probe and build relation for each value d of the join attribute. This choice reduces to the minimum the global redistribution cost.
In the rest of this paper, we use the same threshold frequency as in the redistribution algorithm of *fa_join* [3], i.e. $f_0 = p * log(p)$. For a given value $d \in Hist(R \bowtie S)$,

- the value $index(d) = 0$, means that the frequency of tuples of relations R and S having the value d are less than the threshold frequency. i.e. $Hist(R)(d) < f_0$ and $Hist(S)(d) < f_0$,
- $index(d) = 1$ means that $Hist(R)(d) \geq f_0$ and $Hist(R)(d) \geq Hist(S)(d)$,
- $index(d) = 2$ means that $Hist(S)(d) \geq f_0$ and $Hist(S)(d) > Hist(R)(d)$.

At the end of this step 2.c, the histogram $Hist(R \bowtie S)$ is evenly partitioned between processors into fragments of size $|Hist_i(R \bowtie S)| \leq \min(\frac{|Hist(R)|}{p}, \frac{|Hist(S)|}{p})$ because every one is selected from a $\frac{1}{p}$-th of $Hist(R)$ and of $Hist(S)$ (obtained through hashing of the attribute values).

Every fragment $Hist_i(R \bowtie S)$ is then broadcasted to all processors in time : $Time_{phase2.d} = O\big(g * |Hist(R \bowtie S)| + l\big)$, and the histogram creation has therefore taken the sum of the above steps:
$Time_{phase2} = O\Big((1 + g) * (|Hist(R)| + |Hist(S)|) + l\Big)$.

Note that $Hist(R \bowtie S) \equiv Hist(R) \cap Hist(S)$ and $|Hist(R \bowtie S)|$ is generally very small compared to $|Hist(R)|$ and $|Hist(S)|$.

Phase 3 : Processing local semi-joins

To reduce the redistribution cost and thus the communication costs, we then compute the local semi-joins $\widetilde{R_i}$ and $\widetilde{S_i}$ as follows: $\widetilde{R_i} = R_i \ltimes S$ (resp. $\widetilde{S_i} = S_i \ltimes R$). This is done by a sequential scan of the fragment R_i (resp. S_i), on each processor, by accessing the hash table of $Hist(R \bowtie S)$ in time: $Time_{phase3} = O\Big(\max_{i=1,..,p}(|R_i| + |S_i|)\Big)$.

We observe that, unlike the hash-based algorithms where both relation R and S are redistributed, and as an improvement to the *fa_join* algorithm where only relations R and $S \ltimes R$ are redistributed, we will only redistribute $R \ltimes S$ and $S \ltimes R$ to perform the join operation $R \bowtie S$.

Phase 4 : Creation of communication templates
The attribute values which could lead to attribute value skew (those having high frequencies) are also those which may cause join product skew in standard algorithms. To eliminate this effect, we partition the histogram $Hist(R \bowtie S)$ into two sub-histograms: $\widehat{Hist}(R \bowtie S)$ and $\widetilde{Hist}(R \bowtie S)$ in the following manner: the values $d \in \widehat{Hist}(R \bowtie S)$ are associated to frequencies less than the threshold frequency. The tuples having these values for the join attribute, have no effect neither on AVS nor JPS. These tuples will be redistributed using a hash function. the values $d \in \widetilde{Hist}(R \bowtie S)$ are associated to frequencies greater than the threshold frequency f_0(i.e. $index(d) = 1$ or $index(d) = 2$). Tuples having these join attribute values, have an important effect on attribute value and join product skews. They will be redistributed using an appropriate redistribution algorithm to efficiently avoid both AVS and JPS.

To avoid data skew and thus balance the load among processors, redistribution of the semi-joins $\widetilde{R_i} = R_i \ltimes S$ (resp. $\widetilde{S_i} = S_i \ltimes R$) is necessary.

4.a On each processor i, the relation $\widetilde{R_i} = R_i \ltimes S$ (resp. $\widetilde{S_i} = S_i \ltimes R$) is divided into the three sub-relations $\widetilde{R_i'}$, $\widetilde{R_i''}$ and $\widetilde{R_i'''}$ (resp. $\widetilde{S_i'}$, $\widetilde{S_i''}$ and $\widetilde{S_i'''}$) in the following manner: $\widetilde{R_i} = \widetilde{R_i'} \cup \widetilde{R_i''} \cup \widetilde{R_i'''}$ (resp. $\widetilde{S_i} = \widetilde{S_i'} \cup \widetilde{S_i''} \cup \widetilde{S_i'''}$) where:

- All the tuples of relation $\widetilde{R_i'}$ (resp. $\widetilde{S_i'}$) are associated to values d such that $index(d) = 1$ (resp. $index(d) = 2$),
- All the tuples of relation $\widetilde{R_i''}$ (resp. $\widetilde{S_i''}$) are associated to values d such that $index(d) = 2$ (resp. $index(d) = 1$),
- All the tuples of relation $\widetilde{R_i'''}$ (resp. $\widetilde{S_i'''}$) are associated to values d such that $index(d) = 0$ (resp. $index(d) = 0$), i.e. the tuples which occur with low frequencies in both relations R and S.

This step costs at most : $Time_{phase4.a} = O(\max_{i=1,..,p}(|R_i| + |S_i|))$.

4.b Redistribution of $\widetilde{R_i'}$ and $\widetilde{S_i'}$ 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 the standard algorithms.

To this end, we first create a communication template: the list of messages which constitute the relations' redistribution. This step is performed jointly by all processors, **each one not necessarily computing the list of its own messages, so as to balance the overall process**. This step is completed (see [2]) in time: $Time_{phase4.b} = O(\max_{i=1,..,p} |\widetilde{Hist}_i(R \bowtie S)|)$. Phase 4, has

therefore taken the sum of the above two steps:
$Time_{phase4} = O\Big(\max_{i=1,..,p}(|R_i| + |S_i| + |\widetilde{Hist}_i(R \bowtie S)|)\Big)$.

Phase 5 : Data redistribution

5.a Redistribution of tuples having $d \in \widetilde{Hist}_i(R \bowtie S)$:

Every processor i holds, for every one of its local $d \in \widetilde{Hist}_i(R \bowtie S)$, the non-zero communication volumes it prescribes as a part of communication template. This step (see [2]) costs: $Time_{phase5.a} = O\left(g * |\widetilde{Hist}(R \bowtie S)| + |\widetilde{Hist}(R \bowtie S)| + l\right)$.

5.b Redistribution of tuples with values $d \in \widehat{Hist}_i(R \bowtie S)$:

To redistribute tuples having $d \in \widehat{Hist}_i(R \bowtie S)$, we first broadcast the fragments $\widehat{Hist}_i(R \bowtie S)$ to all processors in : $Time_{phase5.b} = O(g * |\widehat{Hist}(R \bowtie S)| + l)$, because every processor receives the whole $\widehat{Hist}(R \bowtie S)$ which is no more than it sends. At the end of this step, each processor, has local knowledge on tuples which will be redistributed using a hash function. At the end of steps $5.a$ and $5.b$, each processor i, has local knowledge of how the semi joins $\widetilde{R_i}$ and $\widetilde{S_i}$ will be redistributed. Redistribution is then performed, in time: $O(g * (|\widetilde{R_i}| + |\widetilde{S_i}|) + l)$. Phase 5, has therefore taken:
$Time_{phase5} = O\Big(g * (|Hist(R \bowtie S)| + |\widetilde{R_i}| + |\widetilde{S_i}|) + |\widetilde{Hist}(R \bowtie S)| + l\Big)$,
and the complete redistribution algorithm costs $Time_{redist}$:

$$O\Big(\max_{i=1,..,p} (|R_i| + |S_i|) + g * (|\widetilde{R_i}| + |\widetilde{S_i}|) + (1 + g) * (|Hist(R)| + |Hist(S)|) + l\Big).$$

We mention that, we only redistribute the semi-joins $\widetilde{R_i}$ and $\widetilde{S_i}$. Note that $|\widetilde{R_i}|$ (resp. $|\widetilde{S_i}|$) is generally very small compared to $|R_i|$ (resp. $|S_i|$) and $|Hist(R \bowtie S)|$ is generally very small compared to $|Hist(R)|$ and $|Hist(S)|$. Thus we reduce the communication cost to a minimum.

4 Symmetric frequency-adaptive join algorithm

To perform the join of two relations R and S, we first redistribute relations R and S using the above redistribution algorithm at the cost of $Time_{redist} =$

$$O\Big(\max_{i=1,..,p} (|R_i| + |S_i|) + g * (|\widetilde{R_i}| + |\widetilde{S_i}|) + (1 + g) * (|Hist(R)| + |Hist(S)|) + l\Big).$$

Once the redistribution phase is completed, the semi-joins $\widetilde{R_i}$ (resp. $\widetilde{S_i}$) are partitioned into three disjoint relations as follow: $\widetilde{R_i} = \widetilde{R'_i} \cup \widetilde{R''_i} \cup \widetilde{R'''_i}$ (resp. $\widetilde{S_i} = \widetilde{S'_i} \cup \widetilde{S''_i} \cup \widetilde{S'''_i}$) as described in step 4.a of phase 4.
Taking advantage of the identity:

$$R \bowtie S = \Big(\bigcup_i \widetilde{R'_i} \bowtie \widetilde{S''}\Big) \cup \Big(\bigcup_i \widetilde{R''} \bowtie \widetilde{S'_i}\Big) \cup \Big(\bigcup_i \widetilde{R'''_i} \bowtie \widetilde{S'''_i}\Big). \quad (1)$$

Frequencies of tuples of relations $\widetilde{R'_i}$ (resp. $\widetilde{S'_i}$) are by definition greater than the corresponding (matching) tuples in relations $\widetilde{S''_i}$ (resp. $\widetilde{R''_i}$). Fragments $\widetilde{R'_i}$ (resp. $\widetilde{S'_i}$) will be thus chosen as *build* relations and $\widetilde{S''_i}$ (resp. $\widetilde{R''_i}$) as *probe* relations to be duplicated on each processor. This improves over fa_join where all the semi-join $S \ltimes R$ is duplicated. It reduces communications costs significantly in asymmetric cases where both relations contain frequent & infrequent values.

To perform $R \bowtie S$, it is sufficient to compute the three following local joins $\widetilde{R'_i} \bowtie \widetilde{S''}$, $\widetilde{R''} \bowtie \widetilde{S'_i}$ and $\widetilde{R'''_i} \bowtie \widetilde{S'''_i}$ (cf. equation 1). To this end, we broadcast the fragments $\widetilde{R''_i}$ (resp. $\widetilde{S''_i}$) to all processors and complete local joins in all [2] in :

$$Time_{local-join} = O\Big(g * (|\widetilde{R''}| + |\widetilde{S''}|) + \max_{i:1,..,p}(|\widetilde{R'_i} \bowtie \widetilde{S''}| + |\widetilde{R''} \bowtie \widetilde{S'_i}| + |\widetilde{R'''_i} \bowtie \widetilde{S'''_i}|) + l\Big).$$

The global cost of the join of relations R and S using the symmetric frequency-adaptive join (*sfa_join*) algorithm is the sum of the redistribution cost with local join computation cost. It is of the order : $Time_{sfa-join} =$

$$O\Big(\max_{i=1,..,p}(|R_i| + |S_i|) + g * (|\widetilde{R_i}| + |\widetilde{S_i}|) + (1+g) * (|Hist(R)| + |Hist(S)|)$$
$$+ \max_{i:1,..,p}(|\widetilde{R'_i} \bowtie \widetilde{S''}| + |\widetilde{R''} \bowtie \widetilde{S'_i}| + |\widetilde{R'''_i} \bowtie \widetilde{S'''_i}|) + g * (|\widetilde{R''}| + |\widetilde{S''}|) + l\Big).$$

Remark : It can be shown that *sfa_join* algorithm has `optimal` asymptotic complexity when : $\max(|Hist(R)|, |Hist(S)|) \leq \max(\frac{|R|}{p}, \frac{|S|}{p}, \frac{|R \bowtie S|}{p})$,
This inequality often holds, because local joins have almost the same size and the histograms are often very small with respect to the relations.

5 Experimental results

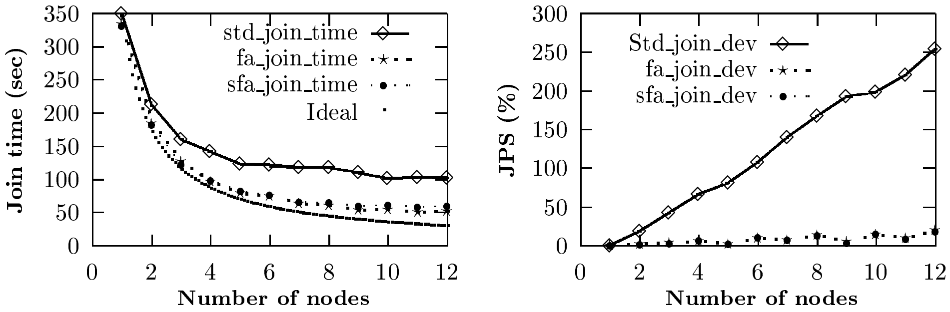
The symmetric frequency-adaptive algorithm (sfa_join) for equi-join has been compared for speed-up and skew with standard hash-join (std_join) and the frequency-adaptive join algorithm (fa_join) introduced in [3]. Tests have used a Fujitsu AP-3000 shared-nothing architecture (based on Sparc Ultra-1 processors with 512Mb of memory) using MySQL-3.22 database servers for local relational processing and MPI-1.1 as message passing interface for communications. Frequencies of join attribute values have been taken to follow the Zipf distribution [16, 6] as it is the case in most database tests.

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[3] has been measured to vary from 0.00% to 0.15% as a result of variations of the join attribute correlation factor.

The speed-up test confirms the absence of a net overhead for sfa_join with respect to fa_join algorithm. 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 both sfa_join and fa_join algorithms which are better than std_join algorithm (Fig. 1.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 sfa_join and fa_join algorithms. This is shown by Fig. 1.b

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

where the JPS is measured by the maximum deviation of the local join result with respect to the average over all processors. As a result, when processing multiple join operations the cost of redistribution will be much smaller with sfa_join and fa_join algorithms than with the std_join algorithm. Figure 1.a shows that



(a) (b)

Fig. 1. Speed_up performance for small attribute skew.

sfa_join is slightly better than *fa_join*, this is mainly due to the redistribution cost which is lower for *sfa_join* compared to *fa_join*. We expect that the gap between the two algorithms will be large when computing very large relations. Currently, our tests are limited by the available disk space.

Two tests were performed to study the effect of attribute value skew on performance. The number of processors has been fixed to 12, relations R and S have been given sizes of $100K$ tuples. The first test used a fixed skew of 0.5 for R while varying skew 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 sfa_join fa_join algorithms are vastly superior to the std_join algorithm while guaranteeing a balanced output. This test shows that sfa_join is slightly better than fa_join algorithm, which confirms its suitability in the presence of any amount of skew (Fig. 2).

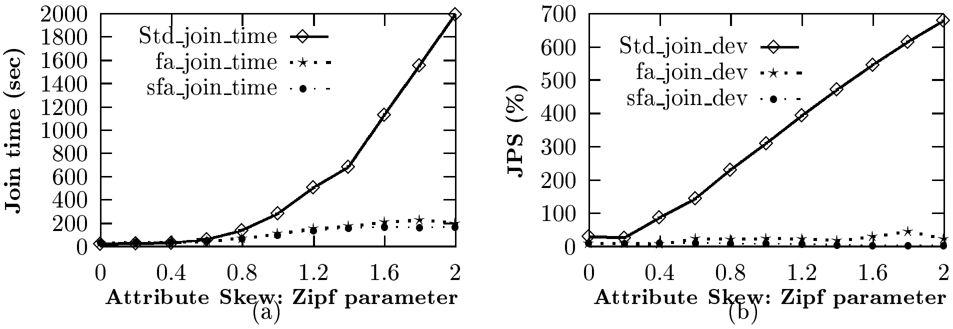


Fig. 2. Effect of attribute value skew for the probe relation.

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. While varying skew factors using Zipf distributions, it is difficult to keep the join result approximately of constant size [4], so the join time in this test is proportional to join result size. Timings in Fig. 3 show that sfa_join algorithm always reaches linear speed-up whereas the timings for fa_join and std_join are proportional to JPS.
Figure 3.b shows that the sfa_join algorithm completely eliminates the problem of JPS. However the *fa_join* fails to avoid join product skew for a Zipf parameter close to 2. This can be explained by the fact that fa_join does not account for cases where a low frequency in the *build* relation combines with a high frequency in the *probe* relation. On the contrary, sfa_join handles the combination of this case and its inverse. The case of high frequencies in both relations is efficiently processed by both algorithms.

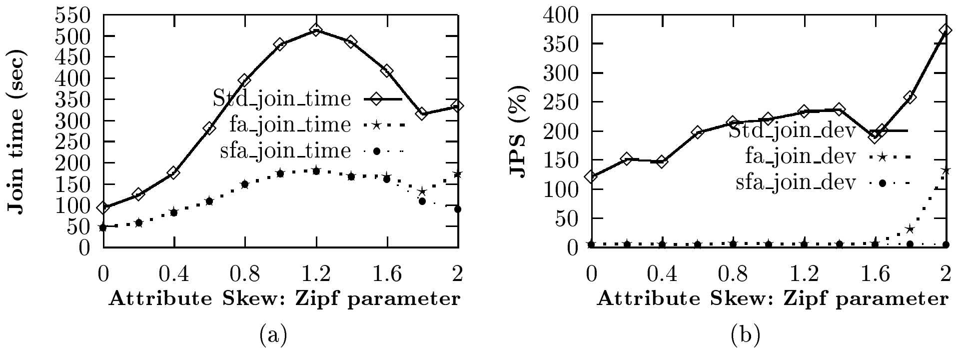


Fig. 3. Effect of high skew for both build and probe relations.

A 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 was fixed to 12, 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%, yielding output relations of sizes between $500K$ and $7M$ tuples. Results [2] show that sfa_join is faster than std_join independently of the join selectivity, while guaranteeing a very small *JPS*. This confirms and strengthens the speed-up test.

6 Conclusion

In this paper, we have introduced an efficient parallel join algorithm based on a "symmetric" sub-set replication allowing to reduce the communication costs while guaranteeing near perfect balancing properties.
The new algorithm is verified to improve on *fa_join* in two ways :1. The negative effect of non-correlated join attribute value distributions is completely eliminated at *no extra cost*. 2. The JPS, already very low for *fa_join*, is now completely eliminated which implies even faster multi-joins. The performance of this algorithm was analyzed using the BSP cost model which predicts a linear speedup.

The $O(\ldots)$ notation only hides small constant factors: they depend only on the implementation but neither on data nor on the BSP machine. The above tests confirm our theoretical predictions about processing overhead, attribute value and join product skews. They also show that sfa_join is never less efficient than fa_join and sometimes more, as predicted by theory.

References

1. M. Bamha and G. Hains. A frequency adaptive join algorithm for SN machines. *Journal of Parallel and Distributed Computing Practices*, 2000. To appear.
2. M. Bamha and G. Hains. A symmetric frequency-adaptive join algorithm for shared nothing machines. Research Report RR-LIFO-2000-03, LIFO, Université d'Orléans, 2000. ftp://ftp-lifo.univ-orleans.fr/pub/RR/RR2000/RR2000-03.ps.
3. M. Bamha and G. Hains. A self-balancing join algorithm for Shared Nothing machines. *In the Proc of the 10th International Conference on Parallel and Distributed Computing Systems, Las Vegas, Nevada*, October 1998.
4. David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshrdri. Practical Skew Handling in Parallel Joins. In *Proceedings of the 18th VLDB Conference, Vancouver, British Columbia, Canada*, 1992.
5. 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.
6. Kian-Lee Tan Hongjun Lu. Dynamic and load-balanced task-oriented database query processing in parallel systems. In *Proceedings of the 3third Conf. Extending Data Base Technology,1992, pp.357-372*, 1992.
7. 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.
8. Hongjun Lu, Beng-Chin Ooi, and Kian-Lee Tan. *Query Processing in Parallel Relational Database Systems*. IEEE Computer Society Press, California, 1994.
9. H. Märtens. Skew-insensitive join processing in shared-disk database systems. *Proc. of Issues and Applications of Database Technology (IADT '98), Berlin*, 1998.
10. 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.
11. Viswanath Poosala and Yannis E. Ioannidis. Estimation of query-result distribution and its application in parallel-join load balancing. *In: Proc. 22th Int. Conf. on Very Large Database Systems, VLDB'96, Bombay, India*, 1996.
12. Donovan A. Schneider and David J. DeWitt. A performance of four parallel join algorithms in a shared-nothing multiprocessor environment. *ACM SIGMOD*, 1989.
13. M. Seetha and Philip S. Yu. Effectiveness of Parallel Joins. *published in the IEEE, Trans. Knowledge and Data Enginneerings, Vol. 2, No 4, pp 410-424*, 1990.
14. Leslie Valiant. A Bridging Model for Parallel Computation,. *Communication of the ACM, Vol 33, No. 8.*, August 1990.
15. Annita N. Wilschut, Jan Flokstra, and Peter M.G. Apers. Parallel Evaluation of Multi-join Queries. *In the Proc. Of the ACM-SIGMOD, California*, 1995.
16. G. K. Zipf. Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology. *Reading, MA, Adisson-Wesley*, 1949.