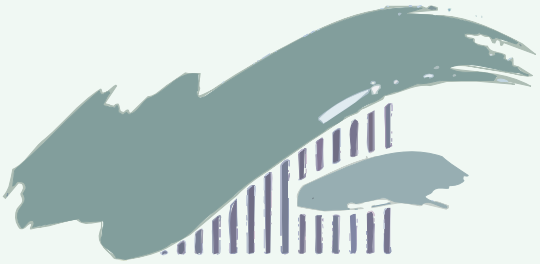

UNIVERSITE D'ORLEANS
Faculté des Sciences
LIFO

Laboratoire d'Informatique Fondamentale d'Orléans
4, rue Léonard de Vinci, BP 6759
F-45067 Orléans Cedex 2
FRANCE

Rapport de Recherche

www : http://www.univ-orleans.fr/SCIENCES/LIFO/

A symmetric frequency-adaptive join algorithm for Shared Nothing machines

M. Bamha and G. Hains
LIFO, Université d'Orléans.
{bamha,ghains}@lifo.univ-orleans.fr

Rapport N^o **2000-03**

A symmetric frequency-adaptive join algorithm for Shared Nothing machines

M. Bamha and G. Hains
LIFO, Université d'Orléans,
BP 6759, 45067 Orléans cedex 2, France.
{bamha,ghains}@lifo.univ-orleans.fr

Abstract. 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. 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 (frequency-adaptive 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 tradeoff between balancing overhead and speedup is analyzed using the scalable and portable BSP[1] cost model which predicts a negligible join product skew and a near-linear speed-up. This prediction is confirmed by a series of tests, some of which show the algorithm's superiority over its predecessor: fa_join.

Key words: *PDBMS*[2], Intra-transaction parallelism, Parallel joins, Multi-joins, 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 applications such as: data-warehousing, decision support and OLAP[3]. Parallelism can greatly increase processing power in such applications [DMT98, BBH99]. However parallelism can only maintain acceptable performance through efficient algorithms realizing complex queries on dynamic, irregular and distributed data. Such algorithms must be designed to fully exploit the processing power of multi-processor machines and the ability to evenly divide load among processors while minimizing local computation and communication costs inherent to multi-processors machines.

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 [BH99, BH98, SY90, KO90, HL91, WDYT94, HTY95, DNSS92, PI96, HK95, MMSY94].

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, PI96]. In particular, the performance of PDBMS has generally been estimated on queries involving one or two join operations only [WFA95]. However the problem of data skew is more acute with multi-joins because the imbalance of intermediate results is unknown during static query optimization [LOT94].

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:

[1] BSP : Bulk-synchronous parallelism.
[2] PDBMS : Parallel Database Management Systems.
[3] OLAP : On-Line Analytical Processing.

- 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[4] and probe[5] 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.

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 reducing communication costs to a minimum.
   (b) an improved version of the redistribution algorithm of *fa-join* [BH98, BH99] 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.

   We analyze and test the sfa_join algorithm for simple join operations. Its suitability for multi-joins follows from the output's near-perfect balancing *at no extra processing cost.*

The organization of this paper is as follows: We first recall the notions of PDBMS, joins operations and load balancing in section 2. Section 3 introduces the scalable and portable BSP cost model. Section 4 presents a new data redistribution approach for parallel join operations. Section 5 presents the sfa_join algorithm and its complexity. Experimental results are detailed in section 6. We then conclude with some suggestions concerning future work in section 7.

2 PDBMS : join operations and load balancing

2.1 PDBMS, join operations and data skew

Join is an expensive and frequently used operation whose parallelization is highly desirable. 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 relations to be joined and $R \bowtie S = R \bowtie (S \ltimes R) = (R \ltimes S) \bowtie (S \ltimes R)$.

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 shared nothing (SN) machines and logical for shared disk (SD) machines [RS95].

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* [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) [SY90, PI96, LOT94]. The former affects immediate performance and the latter affects the efficiency of output or pipelined operations in the case of a multi-join.

[4] Build relation : the relation used to create the hash table.
[5] Probe relation : the relation used to probe the hash table.

2.2 Load balancing in parallel join operations

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

In the first category, 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 large weight. 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. 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 *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 as before 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. We conclude that all existing methods are sensitive to imbalance when applied multiple times because of JPS.

To address this problem, we introduced in [BH98] 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) [BH98], dynamically combining histogram-based balancing with standard hashing methods. The *fa_join* algorithm avoids the slowdown 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 time. 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 scalable BSP cost model which predicts for it a near-linear speedup.

3 The BSP cost model

Bulk-Synchronous Parallelism (BSP) 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 is assumed to 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. The execution time of a superstep

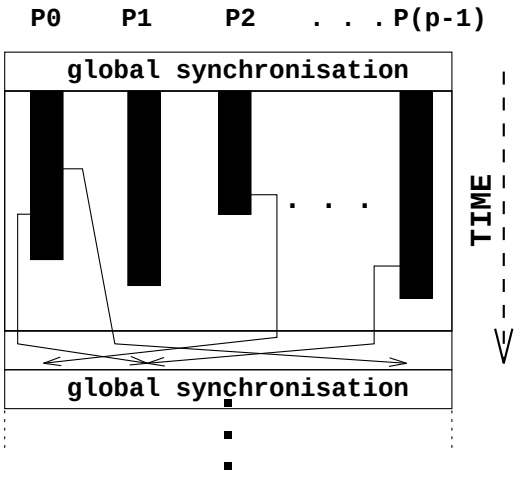

Figure 1: A BSP superstep.

s is thus the sum of the maximal local processing time, of the 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, $\sum_s \text{Time}(s)$, of a BSP program composed of S supersteps is therefore a sum of 3 terms: $W + H * g + S * l$ 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 execution time of a BSP algorithm, design must jointly minimize the number S of supersteps and the total volume h (resp. W) and imbalance $h^{(s)}$ (resp. $W^{(s)}$) of communication (resp. local computation).

4 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 each relation $T \in \{R, S\}$:

- 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 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.

Our redistribution algorithm is the basis for efficient and scalable join 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 :

$$Time_{phase1} = O\left(\max_{i=1,..,p}(|R_i| + |S_i|)\right),$$

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 ⋈ S

To minimize the redistribution cost and thus the communication cost, we compute the histogram of $R \bowtie S$ (note that $Hist(R \bowtie S) \equiv Hist(R) \cap Hist(S)$ in the manner that attribute values which appear in $Hist(R \bowtie S)$ are those which appear in both $Hist(R)$ and $Hist(S)$). This will restrict redistribution to those tuples of relations R and S which will effectively appear in the join result. It thus greatly reduce the global join cost.

The first step is to create the histograms $Hist(R)$ and $Hist(S)$ by a parallel hashing of the histograms $Hist(R_i)$ and $Hist(S_i)$ in time $\max_{i=1,..,p}(|Hist(R_i)| + |Hist(S_i)|)$. The $Hist(R_i)$ 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 : $Time_{phase2.a} = O(g * (|Hist(R)| + |Hist(S)|) + l)$. because in the worst case one processor sends or receives all of the histogram tuples for either R or S.

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* [BH98], 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. $Hist(R)(d) < f_0$ and $Hist(S)(d) < f_0$.
- the value $index(d) = 1$, means that $Hist(R)(d) \geq f_0$ and $Hist(R)(d) \geq Hist(S)(d)$.
- the value $index(d) = 2$, 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 broadcast to all processors in time :

$$Time_{phase2.d} = O\big(g * |Hist(R \bowtie S)| + l\big).$$

This bound holds because every processor receives the complete $Hist(R \bowtie S)$, whose creation has therefore taken the sum of the times of the above steps:

$$\begin{aligned} Time_{phase2} &= Time_{phase2.a} + Time_{phase2.b} + Time_{phase2.c} + Time_{phase2.d} \\ &= O\Big((1+g) * (|Hist(R)| + |Hist(S)|) + l\Big). \end{aligned}$$

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: $\widetilde{R_i} = R_i \ltimes S = R_i \ltimes Hist(R \bowtie S)$ (resp. $\widetilde{S_i} = S_i \ltimes R = S_i \ltimes Hist(R \bowtie S)$). 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 $R \bowtie S$. This will reduce communication costs to a minimum.

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 $index(d) = 2$). The tuples having these values for the join attribute, 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. To this end :

4.a On each processor i, the relation $\widetilde{R_i} = R_i \ltimes S$ is divided into the three sub-relations $\widetilde{R_i'}$, $\widetilde{R_i''}$ and $\widetilde{R_i'''}$ in the following manner : $\widetilde{R_i} = R_i \ltimes S = \widetilde{R_i'} \cup \widetilde{R_i''} \cup \widetilde{R_i'''}$ for $i = 1, .., p$, where :

- All the tuples of relation $\widetilde{R_i'}$ are associated to the values d such that $index(d) = 1$,
- All the tuples of relation $\widetilde{R_i''}$ are associated to the values d such that $index(d) = 2$,
- All the tuples of relation $\widetilde{R_i'''}$ are associated to the values d such that $index(d) = 0$, i.e. the tuples of $\widetilde{R_i}$ which occur with frequencies less than the threshold frequency f_0 (tuples of $\widetilde{R_i'''}$ are associated to low frequencies in both relations R and S).

The cost of this step is at most : $Time_{phase4.a} = O(\max_{i=1,..,p}|R_i|)$.

4.b On each processor i, the relation $\widetilde{S}_i = S_i \ltimes R$ is similarly divided into the three sub-relations $\widetilde{S}'_i$, $\widetilde{S''_i}$ and $\widetilde{S'''_i}$. in the following manner: $\widetilde{S}_i = S_i \ltimes R = \widetilde{S}'_i \cup \widetilde{S''_i} \cup \widetilde{S'''_i}$ for $i = 1, .., p$, where:

- All the tuples of relation $\widetilde{S}'_i$ are associated to the values d such that $index(d) = 2$,
- All the tuples of relation $\widetilde{S''_i}$ are associated to the values d such that $index(d) = 1$,
- All the tuples of relation $\widetilde{S'''_i}$ are associated to the values d such that $index(d) = 0$, i.e. the tuples of $\widetilde{S}_i$ which occur with frequencies less than the threshold frequency f_0 (tuples of $\widetilde{S'''_i}$ are associated to low frequencies in both relations R and S).

The cost of this step is at most : $Time_{phase4.b} = O(\max_{i=1,..,p} |S_i|)$.

4.c 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**.

Processor i computes a set of necessary messages relating to the values d it owns in $\widetilde{Hist}_i(R \bowtie S)$. The communication template is derived from the following mapping, its intended result. For relation $T \in \{\widetilde{R'}, \widetilde{S'}\}$, tuples of T are mapped to multiple nodes as follow:

---

if $\Big(Hist(T)(d) \mod(p) = 0\Big)$ **then**
each processor j will hold an equal number $block_j(d) = \frac{Hist(T)(d)}{p}$ 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 $j_0 + (Hist(T)(d) \mod(p))$ **then**
the processor of index j will hold a block of size : $block_j(d) = \lfloor \frac{Hist(T)(d)}{p} \rfloor + 1$
else
processor of index j will hold a block of size : $block_j(d) = \lfloor \frac{Hist(T)(d)}{p} \rfloor$.

end.

---

where $\lfloor x \rfloor$ is the largest integral value not greater than x and $block_j(d)$ be the number of tuples of value d which processor j should own after redistribution of the fragments T_i of relation T.

The absolute value of $Rest_j(d) = Hist_j(T)(d) - block_j(d)$ 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 $d \in \widetilde{Hist}_i(R \bowtie S)$, processor i owns 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. This information constitutes the communication template. Only those j for which $Rest_j(d) > 0$ (resp. < 0) send (resp. receive) tuples of value d. This step is thus completed in time : $Time_{phase4.c} = O(\max_{i=1,..,p} |\widetilde{Hist}_i(R \bowtie S)|)$.

Phase 4, has therefore taken the sum of the above three steps:

$$\begin{aligned} Time_{phase4} &= Time_{phase4.a} + Time_{phase4.b} + Time_{phase4.c} \\ &= O\Big(\max_{i=1,..,p} (|R_i| + |S_i| + |\widetilde{Hist}_i(R \bowtie S)|)\Big). \end{aligned}$$

Phase 5 : Data redistribution

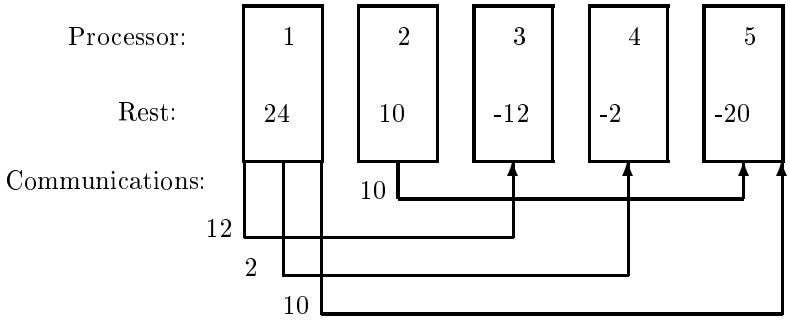

Figure 2: Example of communication template as a function of $Rest$ values.

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: $Rest_j(d) \neq 0$ for $j = 1, .., p$. 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 $Proc(k)_{k=1..\alpha+\beta}$ the array of processor indices for which $Rest_j(d) \neq 0$ in the manner that:

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

A sequential traversal of $Proc(k)_{k=1..\alpha+\beta}$ 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 $O(|\widetilde{Hist}(R \bowtie S)|)$ because for a given d, no more than $(p-1)$ processors can send data and each processor i is in charge of redistribution of tuples having $d \in \widetilde{Hist}_i(R \bowtie S)$. For every i and $d \in \widetilde{Hist}_i(R \bowtie S)$, all the orders `order_to_send`$(j, i, ...)$ are sent to processor j when $j \neq i$ in time $O\left(g * |\widetilde{Hist}(R \bowtie S)| + l\right)$. Figure 2 gives an example of $Rest$ values for a certain d and corresponding sending orders.

In all, this step costs at most:

$$Time_{phase5.a} = O\left(g * |\widetilde{Hist}(R \bowtie S)| + |\widetilde{Hist}(R \bowtie S)| + l\right).$$

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:

$$Time_{phase5.b} = O(g * |\widehat{Hist}(R \bowtie S)| + l),$$

This bound holds 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 tuples of 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:

$$\begin{aligned} Time_{phase5} &= O\Big(g * |\widetilde{Hist}(R \bowtie S)| + |\widetilde{Hist}(R \bowtie S)| \\ &\qquad + g * |\widehat{Hist}(R \bowtie S)| + g * (|\widetilde{R_i}| + |\widetilde{S}_i|) + l\Big) \\ &= O\Big(g * (|Hist(R \bowtie S)| + |\widetilde{R_i}| + |\widetilde{S}_i|) + |\widetilde{Hist}(R \bowtie S)| + l\Big), \end{aligned}$$

and the complete redistribution algorithm costs :

$$\begin{aligned} Time_{redistribution} &= Time_{phase1} + Time_{phase2} + Time_{phase3} + Time_{phase4} + Time_{phase5} \\ &= O\Big(\max_{i=1,..,p}(|R_i| + |S_i|) + (1+g) * (|Hist(R)| + |Hist(S)|) + l \\ &\qquad + \max_{i=1,..,p}(|R_i| + |S_i|) + \max_{i=1,..,p}(|R_i| + |S_i| + |\widetilde{Hist}_i(R \bowtie S)|) \\ &\qquad + g * (|Hist(R \bowtie S)| + |\widetilde{R_i}| + |\widetilde{S}_i|) + |\widetilde{Hist}(R \bowtie S)| + l\Big). \\ &= O\Big(\max_{i=1,..,p}(|R_i| + |S_i|) + g * (|\widetilde{R_i}| + |\widetilde{S}_i|) + (1+g) * (|Hist(R)| + |Hist(S)|) + l\Big). \end{aligned}$$

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.

5 Symmetric frequency-adaptive join algorithm (*sfa_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:

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

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 steps 4.a and 4.b of phase 4.

Taking advantage of the identities:

$$\begin{aligned} R \bowtie S &= \widetilde{R} \bowtie \widetilde{S} = (\widetilde{R'} \cup \widetilde{R''} \cup \widetilde{R'''}) \bowtie (\widetilde{S'} \cup \widetilde{S''} \cup \widetilde{S'''}) \\ &= (\widetilde{R'} \bowtie \widetilde{S''}) \cup (\widetilde{R''} \bowtie \widetilde{S'}) \cup (\widetilde{R'''} \bowtie \widetilde{S'''}) \\ &= (\widetilde{R'} \bowtie \widetilde{S''}) \cup (\widetilde{R''} \bowtie \widetilde{S'}) \cup (\widetilde{R'''} \bowtie \widetilde{S'''}) \\ &= \Big(\bigcup_i \widetilde{R_i'} \bowtie \widetilde{S''}\Big) \cup \Big(\bigcup_i \widetilde{R''} \bowtie \widetilde{S_i'}\Big) \cup \Big(\bigcup_i \widetilde{R_i'''} \bowtie \widetilde{S_i'''}\Big). \end{aligned} \quad (1)$$

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 the join $R \bowtie S$, 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 first broadcast the fragments $\widetilde{R''_i}$ (resp. $\widetilde{S''_i}$) to all processors in time :

$$Time_{step.a} = O\Big(g * (|\widetilde{R''}| + |\widetilde{S''}|) + l\Big).$$

Local joins $\widetilde{R'_i} \bowtie \widetilde{S''}$, $\widetilde{R''} \bowtie \widetilde{S'_i}$ and $\widetilde{R'''_i} \bowtie \widetilde{S'''_i}$ could be done in time :

$$\begin{aligned} Time_{step.b} &= O\Big(\max_{i:1,..,p}(|\widetilde{R'_i}| + |\widetilde{S''}| + |\widetilde{R'_i} \bowtie \widetilde{S''}|) + \max_{i:1,..,p}(|\widetilde{R''}| + |\widetilde{S'_i}| + |\widetilde{R''} \bowtie \widetilde{S'_i}|) \\ &\quad + \max_{i:1,..,p}(|\widetilde{R'''_i}| + |\widetilde{S'''_i}| + |\widetilde{R'''_i} \bowtie \widetilde{S'''_i}|)\Big) \\ &= O\Big(\max_{i:1,..,p}(|\widetilde{R'_i} \bowtie \widetilde{S''}| + |\widetilde{R'_i} \bowtie \widetilde{S''}| + |\widetilde{R'''_i} \bowtie \widetilde{S'''_i}|)\Big). \end{aligned} \quad (2)$$

The equation 2 holds due to the fact that the join size is at least equal to the maximum of the sizes of the semi-joins.

The cost of the local join computation is thus the sum of the two costs above :

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

The global cost of the join of relations R and S using the symmetric frequency-adaptive join algorithm is the sum of the redistribution cost with local join computation cost. It is of the order :

$$\begin{aligned} Time_{sfa-join} &= O\Big(\max_{i=1,..,p}(|R_i| + |S_i|) + g * (|\widetilde{R_i}| + |\widetilde{S_i}|) + (1+g) * (|Hist(R)| + |Hist(S)|) \\ &\quad + \max_{i:1,..,p}(|\widetilde{R'_i} \bowtie \widetilde{S''}| + |\widetilde{R''} \bowtie \widetilde{S'_i}| + |\widetilde{R'''_i} \bowtie \widetilde{S'''_i}|) + g * (|\widetilde{R''}| + |\widetilde{S''}|) + l\Big). \end{aligned}$$

The join of relations R and S using the sfa_join algorithm, 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. This values are mapped to multiple nodes so that local joins have almost the same sizes. 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. We recall that, the redistribution cost is reduced to the minimum because redistribution concerns only tuples which will effectively present in join result.
Note that, local joins have almost the same size $\simeq \frac{|R \bowtie S|}{p}$.

Remark : Observe that sequential θ-join processing of two relations R and S requires at least the following lower bound : $bound_{inf_1} = \Omega\left(|R| + |S| + |R \bowtie_\theta S|\right)$.
Parallel processing with p processors therefore requires $bound_{inf_p} = \frac{1}{p} * \Omega\left(|R| + |S| + |R \bowtie_\theta S|\right)$,
and the *sfa_join* algorithm has optimal asymptotic complexity when :

$$\max(|Hist(R)|, |Hist(S)|) \leq \max(\frac{|R|}{p}, \frac{|S|}{p}, \frac{|R \bowtie S|}{p}), \quad (3)$$

this is due to the fact that, the local joins results have almost the same size and all the terms in $Time_{sfa-join}$ are bounded by those of $bound_{inf_p}$. This inequality often holds, because the histograms are often very small with respect to the relations.

6 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 [BH98]. 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 servers for local relational processing and MPI-1.1[7] as message passing interface for communications.

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 been measured to vary from 0.00% to 0.15% as a result of variations in the correlation factor.

6.1 Speed-up test

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. 3.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. 3.b 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 3.a shows that *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.

6.2 Attribute value skew test

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.

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

[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 T.

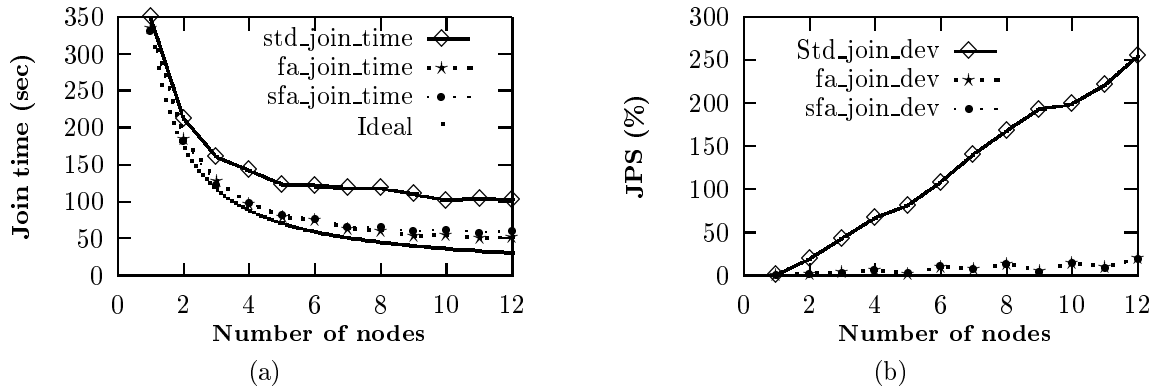

Figure 3: Speed_up performance for small attribute skew.

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. 4).

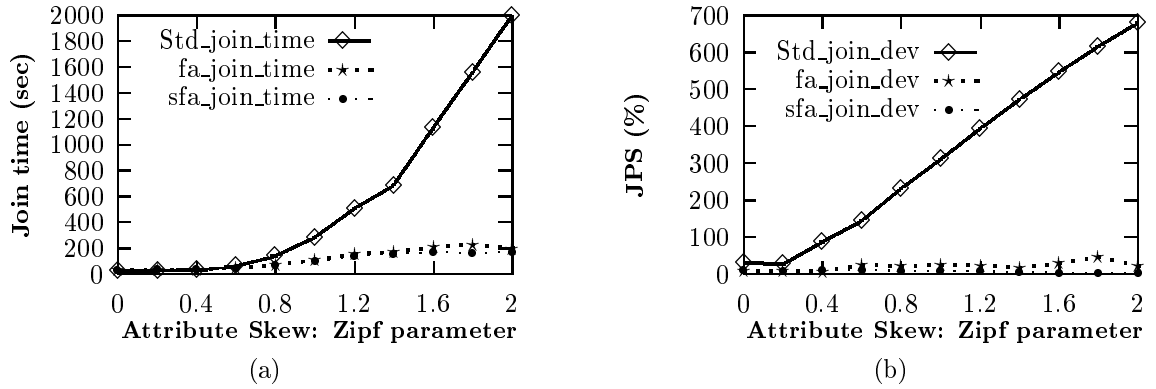

Figure 4: Effect of attribute value skew for the probe relation.

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 [DNSS92], so the join time in this test is proportional to join result size. Timings in Fig. 5 show that sfa_join algorithm always reaches linear speed-up whereas the timings for the fa_join and std_join algorithms are proportional to JPS.

Figure 5.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.

6.3 Join selectivity test

This 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 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 show that fa_join is faster than std_join (Fig. 6.a) independently of the join selectivity, while guaranteeing a very small *JPS* (Fig. 6.b). This confirms and strengthens the results of the speed-up test.

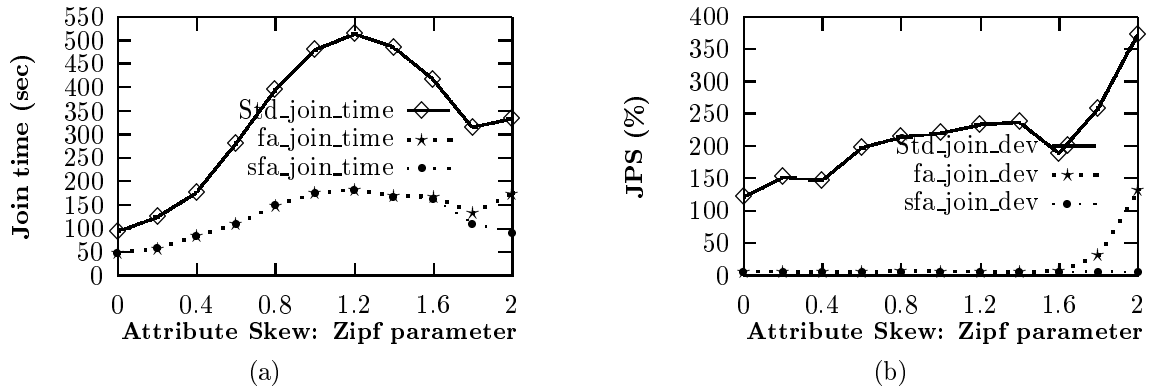

Figure 5: Effect of high skew for both build and probe relations.

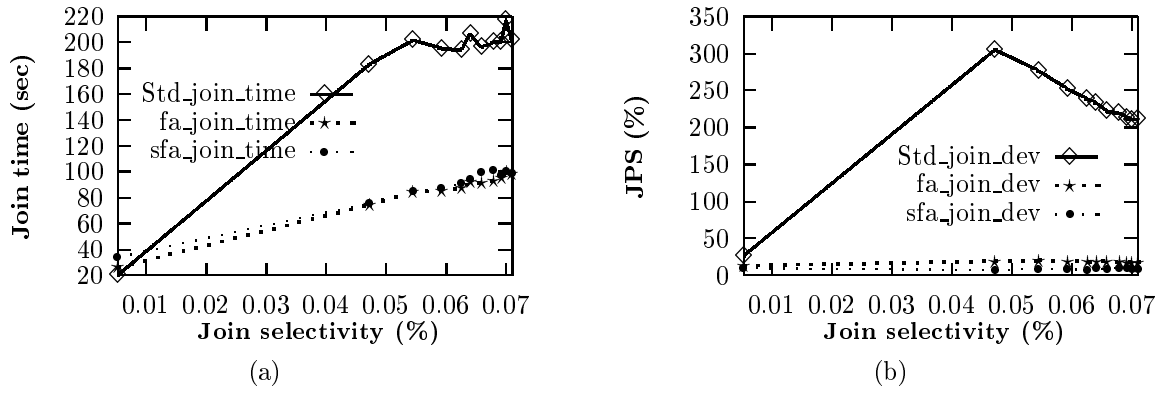

Figure 6: Effect of join selectivity.

7 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. Its predictably low join product skew makes it suitable for multi-join operations.

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 performances of this algorithm were 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 the BSP machine. The above tests confirm our theoretical predictions about processing overhead, attribute value skew and join product skew. They also show that sfa_join is never less efficient than fa_join and sometimes more, as predicted by theory.

Acknowledgment

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

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

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.

[BH99] M. Bamha and G. Hains. A frequency adaptive join algorithm for Shared Nothing machines. *Journal of Parallel and Distributed Computing Practices*, 1999. To appear.

[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.

[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.

[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.

[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.

[SY90] M. Seetha and P. S. Yu. Effectiveness of parallel joins. *IEEE, Transactions on Knowledge and Data Enginneerings*, 2(4):410–424, December 1990.

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

[WDYT94] Joel L. Wolf, Daniel M. Dias, Philip S. Yu, and John Turek. New algorithms for parallelizing relational database joins in the presence of data skew. *IEEE Transactions on Knowledge and Data Engineering*, 6(6):990–997, 1994.

[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.

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