An Optimal Skew-Insensitive Join and Multi-join Algorithm for Distributed Architectures

Mostafa Bamha

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

Abstract. The development of scalable parallel database systems requires the design of efficient algorithms for the join operation which is the most frequent and expensive operation in relational database systems. The join is also the most vulnerable operation to data skew and to the high cost of communication in distributed architectures.

In this paper, we present a new parallel algorithm for join and multi-join operations on distributed architectures based on an efficient semi-join computation technique. This algorithm is proved to have *optimal complexity* and *deterministic perfect load balancing.* Its tradeoff between balancing overhead and speedup is analyzed using the BSP cost model which predicts a negligible join product skew and a linear speed-up. This algorithm improves our *fa_join* and *sfa_join* algorithms by reducing their communication and synchronization cost to a minimum while offering the same load balancing properties even for highly skewed data.

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 (On-Line Analytical Processing). Parallelism can greatly increase processing power in such applications [7,1]. 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 multiprocessor machines and the ability to evenly divide load among processors while minimizing local computation and communication costs inherent to multiprocessor machines. Join is very sensitive to the problem of data skew which can have a disastrous effect on performance [6,4,13,10,9,15,8] due to the high costs of communications and synchronizations in distributed architectures [4,5,2].

Many algorithms have been proposed to handle data skew for join operations [13,10,9]. Such algorithms are not efficient for many reasons:

- the presented algorithms are not scalable (and thus cannot guarantee linear speedup) because their routing decisions are generally performed by a coordinator processor while the other processors are idle,
- they cannot avoid load imbalance between processors because they base their routing decisions on incomplete or statistical information,

K.V. Andersen, J. Debenham, and R. Wagner (Eds.): DEXA 2005, LNCS 3588, pp. 616–625, 2005.

– they cannot solve data skew problem because data redistribution is generally based on hashing data into buckets and hashing is known to be inefficient in the presence of high frequencies.

In this paper we present a new parallel join algorithm called `Osfa_join` (Optimal symmetric frequency adaptive join algorithm) for Shared Nothing machines (i.e architectures where memory and disks are distributed). This algorithm has optimal complexity, perfect balancing properties and supports flexible control of communications induced by intra-transaction parallelism. The `Osfa_join` algorithm is based on an optimal technique for semi-joins computation, presented in [5], and on an improved version of the redistribution algorithm of `sfa-join` [4] which efficiently avoids the problem of attribute value- and join product skews while reducing the communication and synchronization costs to a minimum.

This algorithm guarantees a perfect balancing of the load of the different processors during all the stages of the data redistribution because the data redistribution is carried out jointly by all processors (and not by a coordinator processor). Each processor deals with the redistribution of the data associated to a subset of the join attribute values, not necessarily its "own" values.

The performance of `Osfa_join` is analyzed using the scalable and portable Bulk-synchronous parallel (BSP) cost model [14]. It predicts a negligible join product skew and a linear speed-up, independently of the data and of the (shared nothing) architecture's bandwidth, latency and number of processors.

2 PDBMS, Join Operations and Data Skew

Join is an expensive and frequently used operation whose parallelization is highly desirable. The *join* of two tables or relations R and S on attribute A of R and attribute B of S is the relation, written $R \bowtie S$, containing the pairs of tuples from R and S for which $R.A = S.B$. 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 $R \bowtie S = R \bowtie (S \ltimes R) = (R \ltimes S) \bowtie (S \ltimes R)$.

Parallel join usually proceeds in two phases: a redistribution phase by join attribute hashing and then sequential join of local table 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 redistribution phase) and *join product skew* (imbalance of the output of local joins) [13,11]. The former affects immediate performance and the latter affects the efficiency of output or pipelined multi-join operations.

To address the problem of data skew, we introduced `fa_join` algorithm in [3,6], to avoid the problem of AVS and JPS. However, its performance is suboptimal when computing the join of highly skewed relations because of unnecessary redistribution and communication costs. We introduce here a new parallel algorithm called Osfa_join (Optimal symmetric frequency adaptive join algorithm) to perform such joins. Osfa_join improves on the sfa_join algorithm introduced in [4] by its optimal complexity by using a new approach for semi-joins computation introduced recently in [5]. 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 a linear speedup and an optimal complexity even for highly skewed data.

3 Data Redistribution: An Optimal Approach

In this section, we present an improvement of the algorithm sfa_join [4] called Osfa_join (*Optimal symmetric frequency adaptive join algorithm*) with an optimal complexity. The major difference between sfa_join and Osfa_join lies in the manner of computing semi-joins. We point out that, for semi-joins computation, sfa_join algorithm broadcasts the histograms $Hist_i(R \bowtie S)_{i=1..,p}$ to all processors. Thus, each processor has a local access to the whole histogram $Hist(R \bowtie S)$ to compute local semi-joins. This is not, in general, necessary. In the Osfa_join algorithm, the semi-joins computation is carried out in an optimal way without this stage of broadcast using the techniques presented in [5].

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

- T_i denotes the fragment of relation T placed on processor i,
- $Hist(T)$ denotes the histogram[1] 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 of relation T having the value v for the join attribute,
- $Hist(T_i)$ denotes the histogram of fragment T_i,
- $Hist_i(T)$ is processor i's fragment of the histogram of T,
- $Hist(T)(v)$ is the frequency n_v of value v in relation T,
- $\|T\|$ denotes the number of tuples of relation T, and
- $|T|$ denotes the size (expressed in bytes or number of pages) of relation T.

In the following, we will describe Osfa_join redistribution 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 4 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 scan of the fragment R_i (resp. S_i) on processor i

[1] Histograms are implemented as balanced trees (B-tree): a data structure that maintains an ordered set of data to allow efficient search and insert operations.

in time $c_{i/o} * \max_{i=1,..,p} |R_i|$ (resp. $c_{i/o} * \max_{i=1,..,p} |S_i|$) where $c_{i/o}$ is the cost to read/write a page of data from disk. In principle, this phase costs:

$$Time_{phase1} = O\big(c_{i/o} * \max_{i=1,..,p} (|R_i| + |S_i|)\big),$$

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: Local Semi-joins computation

In order to minimize the redistribution cost and thus the communication time between processors, we then compute the following local semi-joins: $\widetilde{R_i} = R_i \ltimes S$ (resp. $\widetilde{S_i} = S_i \ltimes R$) using proposition 2 presented in [5] in time:

$$Time_{phase2} = O\Big(\min\big(g * |Hist(R)| + \|Hist(R)\|,\ g * \frac{|R|}{p} + \frac{\|R\|}{p}\big) + \max_{i=1,..,p} \|R_i\| \\ + \min\big(g * |Hist(S)| + \|Hist(S)\|,\ g * \frac{|S|}{p} + \frac{\|S\|}{p}\big) + \max_{i=1,..,p} \|S_i\| + l\Big),$$

where g is BSP communication parameter and l the cost of a barrier of synchronisation [14].

We recall (cf. to proposition 1 in [5]) that, in the above equation, for a relation $T \in \{R, S\}$ the term $\min\big(g*|Hist(T)|+\|Hist(T)\|,\ g*\frac{|T|}{p}+\frac{\|T\|}{p}\big)$ is time to compute $Hist_{i=1,...,p}(T)$ starting from the local histograms $Hist(T_i)_{i=1,..,p}$ and during semi-joins computation, we store an extra information called $index(d) \in \{1, 2, 3\}$ for each value $d \in Hist(R \bowtie S)$ [2]. This information will allow 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 permits us to choose dynamically the probe and the build relation for each value d of the join attribute. This choice reduces the global redistribution cost to a minimum.

In the rest of this paper, we use the same threshold frequency as in fa_join algorithm [3,6,4], i.e. $f_0 = p * log(p)$. For a given value $d \in Hist(R \bowtie S)$,

- the value $index(d) = 3$, means that the frequency of tuples of relations R and S, associated to value d, are less than the threshold frequency. $Hist(R)(d) < f_0$ and $Hist(S)(d) < f_0$,
- the value $index(d) = 2$, means that $Hist(S)(d) \geq f_0$ and $Hist(S)(d) > Hist(R)(d)$,
- the value $index(d) = 1$, means that $Hist(R)(d) \geq f_0$ and $Hist(R)(d) \geq Hist(S)(d)$.

Note that, unlike hash-based algorithms where both relation R and S 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.

At the end of this phase, on each processor i, the semi-join $\widetilde{R_i} = R_i \ltimes S$ (resp. $\widetilde{S_i} = S_i \ltimes R$) is divided into three sub-relations in the following way:

$$\widetilde{R_i} = \widetilde{R_i'} \cup \widetilde{R_i''} \cup \widetilde{R_i'''} \quad \text{and} \quad \widetilde{S_i} = \widetilde{S_i'} \cup \widetilde{S_i''} \cup \widetilde{S_i'''} \text{ where:}$$

[2] The size of $Hist(R \bowtie S)$ is generally very small compared to $|Hist(R)|$ and $|Hist(S)|$ because $Hist(R \bowtie S)$ contains only values that appears in both relations R and S.

- 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 relations $\widetilde{R'''_i}$ and $\widetilde{S'''_i}$ are associated to values d such that $index(d) = 3$, i.e. the tuples associated to values which occur with frequencies less than a threshold frequency f_0 in both relations R and S.

Tuples of relations $\widetilde{R'_i}$ and $\widetilde{S'_i}$ are associated to high frequencies for the join attribute. These tuples 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. However the tuples of relations $\widetilde{R'''_i}$ and $\widetilde{S'''_i}$ (are associated to very low frequencies for the join attribute) have no effect neither on AVS nor JPS. These tuples will be redistributed using a hash function.

Phase 3: 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 avoid the slowdown usually caused by attribute value skew and the imbalance of the size of local joins processed by the standard algorithms, an appropriate treatment for high attribute frequencies is needed.

3.a To this end, we partition the histogram $Hist(R \bowtie S)$ into two sub-histograms: $Hist^{(1,2)}(R \bowtie S)$ and $Hist^{(3)}(R \bowtie S)$ in the following manner:

- the values $d \in Hist^{(1,2)}(R \bowtie S)$ are associated to high frequencies of the join attribute (i.e. $index(d) = 1$ or $index(d) = 2$),
- the values $d \in Hist^{(3)}(R \bowtie S)$ are associated to low frequencies of the join attribute (i.e. $index(d) = 3$),

this partition step is performed in parallel, on each processor i, by a local traversal of the histogram $Hist_i(R \bowtie S)$ in time:

$$Time_{3.a} = O(\max_{i=1,..,p} \|Hist_i(R \bowtie S)\|).$$

3.b Communication templates for high frequencies

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 $Hist_i^{(1,2)}(R \bowtie S)$. 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 follows:

```
if $\big(Hist(T)(d) \quad mod(p) = 0\big)$ then
    each processor $j$ will hold : $block_j(d) = \frac{Hist(T)(d)}{p}$ of tuples of value $d$.
else
  begin
    - Pick a random value $j_0$ between 0 and $(p-1)$
    - if $\big($processor index $j$ is between $j_0$ and $j_0 + (Hist(T)(d) \quad mod(p))\big)$ 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 that 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 that processor j must send (if $Rest_j(d) > 0$) or receive (if $Rest_j(d) < 0$).

For $d \in Hist_i^{(1,2)}(R \bowtie S)$, processor i owns a description of the layout of tuples of value d over the network. 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: $Time_{3.b} = O(\|Hist^{(1,2)}(R \bowtie S)\|)$.
The tuples associated to low frequencies (i.e. tuples having $d \in Hist_i^{(3)}(R \bowtie S)$) have no effect neither on the AVS nor the JPS. These tuples are simply mapped to processors using a hash function and thus no communication template computation is needed.

The creation of communication templates has therefore taken the sum of the above two steps:

$$Time_{phase3} = Time_{3.a} + Time_{3.b} = O\big(\max_{i=1,..,p} \|Hist_i(R \bowtie S)\| + \|Hist^{(1,2)}(R \bowtie S)\|\big).$$

Phase 4: Data redistribution

4.a Redistribution of tuples having $d \in Hist_i^{(1,2)}(R \bowtie S)$:

Every processor i holds, for every one of its local $d \in Hist_i^{(1,2)}(R \bowtie S)$, 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 that each processor j will send. The sending orders concerning attribute value d are computed using the following procedure:

```
$i := 1;$    $j := \alpha + 1;$
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 $i := i + 1$; endif
            * if $Rest_{Proc(j)}(d) = 0$ then $j := j + 1$; endif
        end.
```

of complexity $O(\|Hist^{(1,2)}(R \bowtie S)\|)$ 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 $d \in Hist_i^{(1,2)}(R \bowtie S)$.

For each processor i and $d \in Hist_i^{(1,2)}(R \bowtie S)$, all the orders order_to_send$(j, i, ...)$ are sent to processor j when $j \neq i$ in time $O\left(g * |Hist^{(1,2)}(R \bowtie S)| + l\right)$.

In all, this step costs: $Time_{4.\mathrm{a}} = O\big(g * |Hist^{(1,2)}(R \bowtie S)| + \|Hist^{(1,2)}(R \bowtie S)\| + l\big)$.

4.b Redistribution of tuples with values $d \in Hist_i^{(3)}(R \bowtie S)$:

Tuples of relations $\widetilde{R_i'''}$ and $\widetilde{S_i'''}$ (i.e. tuples having $d \in Hist_i^{(3)}(R \bowtie S)$) are associated to low frequencies, they have no effect neither on the AVS nor the JPS. These relations are redistributed using a hash function.

At the end of steps $4.a$ and $4.b$, 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: $Time_{4.\mathrm{b}} = O(g * (|\widetilde{R_i}| + |\widetilde{S_i}|) + l)$.

Phase 4, has therefore taken the sum of the above two costs:

$$Time_{phase4} = O\big(g * \max_{i=1,..,p} (|\widetilde{R_i}| + |\widetilde{S_i}| + |Hist^{(1,2)}(R \bowtie S)|) + \|Hist^{(1,2)}(R \bowtie S)\| + l\big),$$

and the complete redistribution algorithm costs:

$$\begin{aligned} Time_{redist} = O\big(c_{i/o} &* \max_{i=1..p} (|R_i| + |S_i|) + \min\left(g * |Hist(R)| + \|Hist(R)\|, g * \frac{|R|}{p} + \frac{\|R\|}{p}\right) \\ &+ \max_{i=1..p} \|R_i\| + \max_{i=1..p} \|S_i\| + \min\left(g * |Hist(S)| + \|Hist(S)\|, g * \frac{|S|}{p} + \frac{\|S\|}{p}\right) \\ &+ g * (|\widetilde{R_i}| + |\widetilde{S_i}| + |Hist^{(1,2)}(R \bowtie S)|) + \|Hist^{(1,2)}(R \bowtie S)\| + l\big). \qquad (1) \end{aligned}$$

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.

4 Osfa_join : An Optimal Skew-insensitive 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 $Time_{redist}$ (see equation 1 in previous section).

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 phase 2.
Taking advantage of the identities :

$$\begin{aligned} R \bowtie S &= \widetilde{R} \bowtie \widetilde{S} = (\widetilde{R'} \cup \widetilde{R''} \cup \widetilde{R'''}) \bowtie (\widetilde{S'} \cup \widetilde{S''} \cup \widetilde{S'''}) \\ &= (\widetilde{R'} \bowtie \widetilde{S''}) \cup (\widetilde{R''} \bowtie \widetilde{S'}) \cup (\widetilde{R'''} \bowtie \widetilde{S'''}) \\ &= (\widetilde{R'} \bowtie \widetilde{S''}) \cup (\widetilde{R''} \bowtie \widetilde{S'}) \cup (\widetilde{R'''} \bowtie \widetilde{S'''}) \\ &= \Big(\bigcup_i \widetilde{R'_i} \bowtie \widetilde{S''}\Big) \cup \Big(\bigcup_i \widetilde{R''} \bowtie \widetilde{S'_i}\Big) \cup \Big(\bigcup_i \widetilde{R'''_i} \bowtie \widetilde{S'''_i}\Big). \qquad (2) \end{aligned}$$

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 and 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 2). 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 :

$$\begin{aligned} Time_{step.b} = c_{i/o} * O\big(&\max_{i:1,..,p}(|\widetilde{R'_i}| + |\widetilde{S''}| + |\widetilde{R'_i} \bowtie \widetilde{S''}|) + \max_{i:1,..,p}(|\widetilde{R''}| + |\widetilde{S'_i}| + |\widetilde{R''} \bowtie \widetilde{S'_i}|) \\ &+ \max_{i:1,..,p}(|\widetilde{R'''_i}| + |\widetilde{S'''_i}| + |\widetilde{R'''_i} \bowtie \widetilde{S'''_i}|)\big) \\ = c_{i/o} * O\big(&\max_{i:1,..,p}(|\widetilde{R'_i} \bowtie \widetilde{S''}| + |\widetilde{R'_i} \bowtie \widetilde{S''}| + |\widetilde{R'''_i} \bowtie \widetilde{S'''_i}|)\big). \qquad (3) \end{aligned}$$

The equation 3 holds due to the fact that the join size is at least equal to the maximum of the semi-joins sizes.
The cost of the local join computation is thus the sum of the two costs above :

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

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

$$Time_{Osfa\_join} = O\Big(c_{i/o} * \max_{i=1,..,p}(|R_i| + |S_i|) + \max_{i=1,..,p}\|R_i\| + \max_{i=1,..,p}\|S_i\| + l$$

$$
\begin{aligned}
&+ \min\big(g * |Hist(R)| + \|Hist(R)\|,\ g * \frac{|R|}{p} + \frac{\|R\|}{p}\big) \\
&+ \min\big(g * |Hist(S)| + \|Hist(S)\|,\ g * \frac{|S|}{p} + \frac{\|S\|}{p}\big) \\
&+ g * (|\widetilde{R_i}| + |\widetilde{S_i}| + |Hist^{(1,2)}(R \bowtie S)|) + \|Hist^{(1,2)}(R \bowtie S)\| \\
&+ g * (|\widetilde{R''}| + |\widetilde{S''}|) + c_{i/o} * \max_{i:1,..,p} (|\widetilde{R'_i} \bowtie \widetilde{S''}| + |\widetilde{R''} \bowtie \widetilde{S'_i}| + |\widetilde{R'''_i} \bowtie \widetilde{S'''_i}|)\Big).
\end{aligned}
$$

The join of relations R and S using the Osfa_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 minimal because redistribution concerns only tuples which are effectively present in the join result.
Note that, the size of $\widetilde{R''}$ (resp. $\widetilde{S''}$) est generally very small compared to the size of the join $\widetilde{R''} \bowtie \widetilde{S'_i}$ (resp. $\widetilde{R'_i} \bowtie \widetilde{S''}$) and the size of local join on each processor i, $|\widetilde{R'_i} \bowtie \widetilde{S''}| + |\widetilde{R''} \bowtie \widetilde{S'_i}| + |\widetilde{R'''_i} \bowtie \widetilde{S'''_i}|$, have almost the same size $\simeq \frac{|R \bowtie S|}{p}$.

Remark : Sequential join processing of two relations R and S requires at least the following lower bound : $bound_{inf_1} = \Omega\left(c_{i/o} * (|R| + |S| + |R \bowtie S|)\right)$.
Parallel processing with p processors requires therefore : $bound_{inf_p} = \frac{1}{p} * bound_{inf_1}$. and `Osfa_join` algorithm has *optimal* asymptotic complexity when :

$$
|Hist^{(1,2)}(R \bowtie S)| \leq c_{i/o} * \max(\frac{|R|}{p}, \frac{|S|}{p}, \frac{|R \bowtie S|}{p}) \qquad (4)
$$

this is due to the fact that, the local joins results have almost the same size and all the terms in $Time_{Osfa-join}$ are bounded by those of $bound_{inf_p}$. This inequality holds, if we choose a threshold frequency f_0 greater than p (which is the case for our threshold frequency $f_0 = p * log(p)$).

5 Conclusion

In this paper, we have introduced the first parallel join algorithm with `optimal complexity` based on an efficient semi-join algorithm introduced in [5] and on a "symmetric" sub-set replication technique allowing to reduce the communication costs to the minimum while guaranteeing near perfect balancing properties. The algorithm `Osfa_join` is proved to have an optimal complexity even in the presence of highly skewed data. Its predictably low join product skew makes it suitable for multi-join operations.

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. Our experience with the join operation [4,6,3,2] is evidence that the above theoretical analysis is accurate in practice.

References

1. 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*, LNCS 1677, pages 616–625, Florence, Italy, 1999. Springer-Verlag.
2. M. Bamha and M. Exbrayat. Pipelining a skew-insensitive parallel join algorithm. *Parallel Processing Letters*, Volume 13(3), pages 317–328, 2003.
3. M. Bamha and G. Hains. A self-balancing join algorithm for SN machines. *Proceedings of International Conference on Parallel and Distributed Computing and Systems (PDCS)*, pages 285–290, Las Vegas, Nevada, USA, October 1998.
4. M. Bamha and G. Hains. A skew insensitive algorithm for join and multi-join operation on Shared Nothing machines. In *the 11th International Conference on Database and Expert Systems Applications DEXA'2000*, LNCS 1873, 2000.
5. M. Bamha and G. Hains. An efficient equi-semi-join algorithm for distributed architectures. In *the 5th International Conference on Computational Science (ICCS'2005). 22-25 May, Atlanta, USA*, LNCS 3515, pages 755–763, 2005.
6. M. Bamha and G. Hains. A frequency adaptive join algorithm for Shared Nothing machines. *Journal of Parallel and Distributed Computing Practices (PDCP), Volume 3, Number 3, pages 333-345*, September 1999.
7. 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.
8. 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.
9. K. A. Hua and C. Lee. Handling data skew in multiprocessor database computers using partition tuning. In *Proc. of the 17th International Conference on Very Large Data Bases*, pages 525–535, Barcelona, Catalonia, Spain, 1991. Morgan Kaufmann.
10. 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.
11. 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.
12. D. Schneider and D. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *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.
13. M. Seetha and P. S. Yu. Effectiveness of parallel joins. *IEEE, Transactions on Knowledge and Data Enginneerings*, 2(4):410–424, December 1990.
14. D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.
15. 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.