# PIPELINING A SKEW-INSENSITIVE PARALLEL JOIN ALGORITHM

M. BAMHA and M. EXBRAYAT*
*LIFO, Université d'Orléans,*
*BP 6759, 45067 Orléans cedex 2, France.*

ABSTRACT

Most standard parallel join algorithms try to overcome data skews with a relatively static approach. The way they distribute data (and then computation) over nodes depends on a data re-distribution algorithm (hashing or range partitioning) that is determined before the actual join begins. On the contrary, our approach consists in pre-scanning data in order to choose an efficient join method for each given value of the join attribute. This approach has already proved to be efficient both theoretically and practically in our previous papers. In this paper we introduce a new pipelined version of our frequency adaptive join algorithm. The use of pipelining offers flexible strategies for resource allocation while avoiding unnecessary disk input/output of intermediate join result when computing multi-join queries. We present a detailed version of the algorithm and a cost analysis based on the BSP (Bulk Synchronous Parallel) model, showing that our pipelined algorithm achieves noticeable improvements compared to the sequential parallel version for multi-join queries while guaranteeing perfect balancing properties.
*Keywords*: Parallel Data Base Management Systems (PDBMS), Parallel joins, Multi-joins, Data skew, Join-product skew, Pipelined parallelism, Dynamic load balancing.

## 1. Introduction

Parallelism has been successfully introduced within Data Base Management Systems (DBMS) for several years, in order to handle the ever-growing sources of data in many domains such as large transactional (on-line) databases, data-warehousing, On-Line Analytical Processing, decision support and data mining [1]. Nevertheless, parallelizing such operations as *join* still constitutes an active research topic. 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 [2,3] owing to the high costs of communication and synchronization in this architecture. In order to compute a join in parallel, data is first distributed amongst nodes according to a given method (most of time, a hash function). All tuples with a given value of the join attribute will be stored and computed on the same node, and skewed data will thus lead to overloaded processors. As a consequence, many join algorithms have been proposed to handle data skew.

A well known strategy consists in using several nodes to compute a skewed

*{Mostafa.Bamha,Matthieu.Exbrayat}@lifo.univ-orleans.fr

value. This can be done either statically (matching tuples are grouped in small buckets which can be computed on various nodes) or dynamically (through work stealing for instance). These strategies are not fully satisfying. They especially imply extra communication (some data must either be duplicated or re-distributed from an overloaded node to an available one).

On the contrary, the join algorithms we presented in [4,5,6] handle data skew in a much different manner. Data distribution templates are computed in order to avoid the effect of data skew while limiting data duplication as much as possible. To achieve this goal, we use a total data-distribution information in the form of histograms. Tables that have to be joined are first scanned in order to detect skews. Data distribution is then computed on a per-value basis, some tuples being joined according to a parallel hash algorithm, some other being joined according to a duplicate/distribute method. The parallel cost model we apply allows us to guarantee that histogram management has a negligible cost when compared to the gains provided by the reduction of the communication costs and the avoidance of load imbalance among processors. These algorithms guarantee a perfect balancing of the load over the processors during all the stages of the data redistribution because this 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.

Is this paper we propose to extend our algorithms to handle multi-join queries. While a lot of work has been conducted for simple join, little is known for the case of complex queries leading to multi-joins [7,8,9]. In particular, the performance of PDBMS has generally been estimated on queries involving one or two join operations only [10]. However the problem of data skew is more acute with multi-joins. A join algorithm that handles data skew might anyway produce skewed data, and the imbalance of these intermediate results can hardly be estimated during a static query optimization [11].

In this paper, we show that the `Osfa_join` algorithm we introduced in [6], can be efficiently pipelined to handle multi-join queries owing to the fact that this algorithm reduces the join product skew and thus helps to balance the computation of histograms and the redistribution of data. Pipelining offers flexible resource allocation and reduces disk input/output for intermediate results. To allow pipelining, we propose to extend the histograms to handle the values of consecutive join attributes.

The organization of this paper is as follows: we first recall the notions of PDBMS, join operations and load balancing in section 2. We then present parallel execution strategies for multi-queries in section 3, and propose and discuss the use of a pipelined strategy for `Osfa_join` in section 4. We finally conclude in section 5.

## 2. PDBMS: join operations and load balancing

Join is an expensive and frequently used operation whose parallelization is highly desirable. The *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 $R \bowtie S$, containing the pairs of tuples from $R$ and $S$ verifying $R.A = S.B$. 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 $R \bowtie S = R \bowtie (S \ltimes R) = (R \ltimes S) \bowtie (S \ltimes R)$.

In PDBMS, relations are generally partitioned among processors by *horizontal fragmentation* according to the values of a given attribute [12]. Three methods are used: hash partitioning, block-partitioning (also called range partitioning) and cyclic (also called round-robin) partitioning.

Parallel join usually proceeds in two phases: a redistribution phase according to the values of the join attribute and then a sequential join of local table fragments.

Many algorithms have been proposed [8,9,13,14]. 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 main disadvantage is that they are 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) [11]. The former affects immediate performance and the latter affects the efficiency of output of pipelined operations in the case of a multi-join.

To address this problem, we introduced a deterministic data redistribution algorithm with near-perfect balancing properties [4], which is 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 [15,16], leading to general predictions about the effect of relation histograms on performance. The analysis suggests a hybrid *frequency-adaptive* algorithm (`fa_join` algorithm), dynamically combining histogram-based balancing with standard hashing methods. The `fa_join` algorithm avoids the slowdown usually caused by attribute value skew 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 never penalizes overall performance, even in the absence of skew.

The `fa_join` algorithm was thus designed to avoid efficiently the problem of AVS and JPS. However, its performance is sub-optimal when computing the join of highly skewed relations because of unnecessary redistribution and communication costs. We introduced in [6] 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 [5] by its optimal complexity and the semi-joins computation. 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. Parallel execution strategies for multi-join queries

Parallel execution of multi-join queries depends on the execution plan of the simple joins that compose the multi-join. The main difference between these strategies lies in the manner of allocating the simple joins to different processors and in the choice of an appropriate degree of parallelism (i.e. the number of processors used to compute each simple join).

Several strategies were proposed for the evaluation of multi-join queries[10]. These strategies depend generally on the parallel query execution plan. In these strategies both intra-operator and inter-operator parallelism can be used. These strategies are divided into four principal categories presented thereafter.

### *3.1. Sequential parallel execution*

*Sequential parallel execution* is the simplest strategy for evaluating, in parallel, a multi-join query. This strategy does not require inter-operator parallelism since simple joins of the query are evaluated one after the other in a parallel way. This strategy does not induce any pipelined parallelism owing to the fact that a simple join cannot be started until all its operands are entirely available. The execution time of each join is then the execution time of the slowest processor. To reach acceptable performance, the join algorithms used in this strategy should reduce the load imbalance between all the processors.

`Osfa_join` can be used efficiently in this strategy to reduce the idle time, owing to the fact that `Osfa_join` is insensitive to attribute-value and join-product skews and this algorithm has been proved to guarantee a linear speedup.

### *3.2. Parallel synchronous execution*

*Parallel synchronous execution* uses, in addition to intra-operator parallelism, inter-operator parallelism [17,18]. In this strategy several simple join operations can be computed simultaneously on disjoint sets of processors.

The main difficulty in a *parallel synchronous execution* lies in the manner of allocating the simple joins to the available processors and in the choice of an appropriate degree of parallelism to be used for each join. In parallel synchronous strategies, the objective of such an allocation is to reduce the latency so that:

- the global execution time of each operator should be of the same order to avoid the latencies induced by the imbalance in computations of the different groups of processors, and

- the global execution time of each operator in the same group (or set) of processors must also be of the same order for an efficient use of the machine resource (i.e. local computation within each set of processors must be balanced) because the execution time for each operator is the execution time of slowest processor in each group of processors.

The load balancing between the groups of processors depends on the effectiveness of the dynamic balancing strategy used at the time of the query optimization, whereas the load balance of the processors in the same group can be carried out by using the join algorithms we introduced.

### *3.3. Segmented right-deep execution*

Contrary to a parallel synchronous strategy, a *segmented right-deep execution* [19] uses, in addition to intra-operator parallelism, pipelined inter-operator parallelism. Pipelined inter-operator parallelism is used in the evaluation of the right-branches of the query tree.

`Osfa_join` algorithm can be used efficiently in a segmented right-deep execution for the same reasons as for a synchronous execution. However its performance can be increased by alternating the join computation and the communication of the generated tuples. Alternating these two phases helps to avoid massive synchronizations between the various groups of processors after the generation of each tuple, thus reducing the number of synchronization barriers by gathering the data exchanged between the processors, in few messages without disturbing the execution of the joins.

### *3.4. Full parallel execution*

*Full parallel execution* [10] uses inter-operator parallelism and pipelined inter-operator parallelism in addition to intra-operator parallelism in the evaluation of simple joins. In this strategy, all the simple joins, that compose the multi-join query, are computed simultaneously in parallel using disjoint set of processors. Inter-operator parallelism and pipelined inter-operator parallelism are exploited according to the type of the query tree.

A full parallel execution is simply the combination of preceding strategies where all the forms of parallelism (intra-operator parallelism, inter-operator and pipelined inter-operator parallelisms) are exploited. The effectiveness of such a strategy depends on the quality of the execution plans generated during the query optimization phase.

In the following section we will present a detailed approach to introduce pipelining in the `Osfa_join` algorithm.

## 4. Pipelining Osfa_join algorithm

### *4.1. Principle*

Pipelining has been fully studied and successfully implemented with many classical join algorithms. Nevertheless, these algorithms are generally based on hash join techniques and are thus sensitive to attribute-value and join-product skews, which are very likely to occur in a multi-join query. For this reason we propose to adapt `Osfa_join` (which is insensitive to data skew) to pipelined joins. This task in fact appears to be difficult.

Most pipeline strategies use a predefined, static execution plan. All these strategies can be applied only to segmented right-deep join queries. The way data will be distributed over processors for each operator is fixed at query compilation time. When one processor produces a result tuple, this latter can be immediately transmitted to and used by the next operator, which does not need to have a global view of the whole set of manipulated tuples. However, data-skew can have a disastrous effect on the performance of the execution of repeated consecutive join operators.

On the contrary, `Osfa_join` is an adaptive, dynamic operator and the definitive distribution of data is determined at execution time. `Osfa_join` handles pipelining through the use of a multi-attribute pre-scan strategy. This strategy consists in evaluating the correlation between the join attributes of two successive joins. As a consequence, we can forecast the frequencies of intermediate relations, and then the way they should be redistributed for the next join, before their actual computation occurs. Hence we build the histograms for intermediate relations before their tuples are produced, and we do not need to re-scan the results from the disk.

### *4.2. Detailed algorithm*

We first assume that each relation $T_i$ is horizontally fragmented among $p$ processors (which are numbered from 1 to $p$) and:

- $T_{i,j}$ denotes the fragment (sub-relation) of relation $T_i$ placed on processor $j$,
- $Hist^x(T_i)$ denotes the histogram of relation $T_i$ with respect to the join attribute $x$, 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 $x$. The histogram is often much smaller and never larger than the relation it describes,
- $Hist^x(T_{i,j})$ denotes the histogram of fragment $T_{i,j}$ placed on processor $j$,
- $Hist_j^x(T_i)$ is processor $j$'s fragment of the global histogram of relation $T_i$,
- $Hist^x(T_i)(v)$ is the frequency $(n_v)$ of value $v$ in relation $T_i$,
- $Hist^x(T_{i,j})(v)$ is the frequency of value $v$ in sub-relation $T_{i,j}$,
- $\|T_i\|$ denotes the number of tuples of relation $T_i$, and
- $|T_i|$ denotes the size (expressed in bytes or number of pages) of relation $T_i$.

In this section we introduce a parallel pipelined strategy for the `Osfa_join` algorithm applied to a segmented right-deep multi-join query of the form $R_i \bowtie S_i$, where $R_i$ is a source relation and $S_i$ is either a source relation (which should only occur for the first join) or an intermediate result. The execution tree consists of $n$ join operators. The final result is stored as $S_{n+1}$.

Contrary to "classical" pipelined strategies, the pipelined `Osfa_join` can be easily generalized to handle not only segmented right-deep join queries but also more general bushy trees. Algorithm 1 describes the various steps of a pipelined execution of `Osfa_join` algorithm for an n-join segmented right-deep query. We will also give an upper bound of the BSP execution time of each step. The notation O(...) hides only small constant factors: they depend only on the program implementation but neither on data nor on the BSP machine parameters.

**Algorithm 1** Pipelined Osfa_join algorithm

**For** i $= 1$ **to** $n$ **do**
 **Par** (on each node) $j \in [1, p]$ **do**
  **If** i=1 (first join) **Then**
   ▷ Create local histograms $Hist^{a_i}(R_{i,j})$ of relation $R_{i,j}$ on join attribute $a_i$
   ▷ Hash $Hist^{a_i}(R_{i,j})$ to other nodes, and collect all $Hist^{a_i}(R_{i,k})$, $k \neq j$
   ▷ Build the corresponding global histogram fragment $Hist_j^{a_i}(R_i)$ of $R_i$
   ▷ Create local histograms $Hist^{a_i}(S_{i,j})$ of relation $S_{i,j}$ on join attribute $a_i$
   ▷ Hash $Hist^{a_i}(S_{i,j})$ to other nodes, and collect all $Hist^{a_i}(S_{i,k})$, $k \neq j$
   ▷ Build the corresponding global histogram fragment $Hist_j^{a_i}(S_i)$ of $S_i$
  **Else**
   ▷ Hash $Hist^{a_i}(S_{i,j})$ to other nodes, and collect all $Hist^{a_i}(S_{i,k})$, $k \neq j$
   ▷ Build the corresponding global histogram fragment $Hist_j^{a_i}(S_i)$ of $S_i$
  **EndIf**
  ▷ Create global histogram fragments $Hist_j^{a_i}(R_i \bowtie S_i) = Hist_j^{a_i}(S_{(i+1)})$
   of the join result
  ▷ Create communication templates for the values in $Hist_j^{a_i}(S_i)$ and $Hist_j^{a_i}(R_i)$
  ▷ Hash (and collect) the communication templates
  ▷ Exchange data across nodes according to the communication template
  **If** $i < n$ (not the last join) **Then**
   ▷ Create local histograms $Hist^{a_{(i+1)}}(R_{(i+1),j})$ of relation $R_{(i+1),j}$ on $a_{(i+1)}$
   ▷ Hash $Hist^{a_{(i+1)}}(R_{(i+1),j})$ to other nodes and collect all $Hist^{a_{(i+1)}}(R_{(i+1),k})$, $k \neq j$
   ▷ Build the corresponding global histogram fragment $Hist_j^{a_{(i+1)}}(R_{(i+1)})$ of $R_{(i+1)}$
   ▷ Create local histograms $Hist^{a_{(i+1)}}(S_{(i+1),j})$ of relation $S_{(i+1),j}$ on $a_{(i+1)}$
    using histograms and communication templates
  **EndIf**
  ▷ Compute the join $S_{(i+1),j} = (R_{i,j} \bowtie S_{i,j})$
 **EndPar**
**EndFor**

Algorithm 1 can be divided, for each join $R_i \bowtie S_i$, into five phases as follow:

**Phase a: Creating histograms of relations $R_1$ and $S_1$ of the first join.** The histogram $Hist^{a_1}(R_1)$ of relation $R_1$ is created in three steps:

1. Local histograms creation step: local histograms $Hist^{a_1}(R_{1,j})$ of relation $R_1$ are created in parallel, on each processor $j$ by a sequential scan of relation $R_{1,j}$ in time $c_{i/o} * \max_{j=1,\ldots,p} |R_{1,j}|$ where $c_{i/o}$ is a constant factor for disk input/output operations. The global cost of this step, is therefore:

$$Time_{a.1}(R_1) = O\Big(c_{i/o} * \max_{j=1,\ldots,p}(|R_{1,j}|)\Big) \tag{1}$$

2. Hashing of histograms step: at the end of the local histograms creation step, the local histograms $Hist^{a_1}(R_{1,j})$ are hashed to processors using a common hashing function in time:

$$Time_{a.2}(R_1) = O\bigg(\min\Big(g * |Hist^{a_1}(R_1)|, \max_{j=1,\ldots,p} g * |R_{1,j}|\Big) + l\bigg) \tag{2}$$

The term $g * |Hist^{a_1}(R_1)|$ is due to the fact that each processor receives at most a block of the size of the whole histogram of $R_1$. Parameters $g$ and $l$ are respectively the BSP costs for communication and synchronization barrier (see [4,5,6]).

3. Global histogram creation step: starting from the received blocks, the global histogram of relation $R_1$ is created in time:

$$Time_{a.3}(R_1) = O\Big( \min\big(\|Hist^{a_1}(R_1)\|, \max_{j=1,\ldots,p} \|R_{1,j}\|\big)\Big) \tag{3}$$

The global cost of the creation of $Hist^{a_1}(R_1)$ is then:

$$Time_a(R1) \quad = \quad Time_{a1}(R_1) + Time_{a2}(R_1) + Time_{a3}(R_1) \tag{4}$$

Similarly for relation $S_1$, the above technique is used to compute the global histogram $Hist^{a_1}(S_1)$ in time $Time_a(S1)$, which is obtained by replacing $R_1$ by $S_1$ in equation 4. Hence the global cost of the histograms $Hist^{a_1}(R_1)$ and $Hist^{a_1}(S_1)$ creation phase is:

$$\begin{aligned} Time_a &= Time_a(R1) + Time_a(S1) \\ &= O\Big( \min\big((g * |Hist^{a_1}(R_1)| + \|Hist^{a_1}(R_1)\|), \max_{j=1,\ldots,p} g * (|R_{1,j}| + \|R_{1,j}\|)\big) \\ &\quad + \min\big((g * |Hist^{a_1}(S_1)| + \|Hist^{a_1}(S_1)\|), \max_{j=1,\ldots,p} g * (|S_{1,j}| + \|S_{1,j}\|)\big) \\ &\quad + c_{i/o} * \max_{j=1,\ldots,p} |R_{1,j}| + c_{i/o} * \max_{j=1,\ldots,p} |S_{1,j}| + l\Big) \end{aligned}$$

**Phase b: Hashing and creating histograms of relation $S_i$.** After the creation of local histograms $Hist^{a_i}(S_{i,j})$ of relation $S_i$, on each processor, these histograms are hashed to processors using a common hashing function (cf. equation 2) in time:

$$Time_{b.1} \quad = \quad O\Big( \min\big(g * |Hist^{a_i}(S_i)|, \max_{j=1,\ldots,p} g * |S_{i,j}|\big) + l\Big)$$

At the end of the hashing histograms step, the received blocks are merged to compute, on each processor $j$, a fragment $Hist_j^{a_i}(S_i)$ of the global histogram of relation $S_i$ (cf. equation 3) in time:

$$Time_{b.2} \quad = \quad O\Big( \min\big(\|Hist^{a_1}(S_i)\|, \max_{j=1,\ldots,p} \|S_{i,j}\|\big)\Big)$$

The global cost of Phase b is therefore the sum of the above two costs:

$$Time_b = Time_{b.1} + Time_{b.2}$$

**Phase c: Creating communication templates and data redistribution of $R_i$ and $S_i$.** To reduce the communication templates cost, the communication templates are computed for only tuples of relations $R_i$ and $S_i$ that will be present in the join result $S_{(i+1)} = R_i \bowtie S_i$. To this end, we compute the global histogram $Hist_j^{a_i}(R_i \bowtie S_i) = Hist_j^{a_i}(S_{(i+1)})$ of the join result by "intersecting", on each processor $j$, $Hist_j^{a_i}(R_i)$ and $Hist_j^{a_i}(S_i)$ (cf. [4,5,6]) in time:

$$Time_{\mathrm{c.1}} = O\Big(\max_{j=1,\ldots,p}(\min(\|Hist_j^{a_i}(R_i)\|, \|Hist_j^{a_i}(S_i)\|))\Big)$$

Owing to the fact that tuples with high frequencies are often tuples which could lead to attribute value and join product skews, an appropriate treatment for high frequencies is needed to avoid the slowdown caused by attribute value skew and the imbalance of the size of local joins. To this end, the histogram $Hist_j^{a_i}(R_i \bowtie S_i)$[†]is partitioned into two sub-histograms: $\widehat{Hist}_j^{a_i}(R_i \bowtie S_i)$ and $\widetilde{Hist}_j^{a_i}(R_i \bowtie S_i)$ so that

- the values $v \in \widehat{Hist}_j^{a_i}(R_i \bowtie S_i)$ are associated to low frequencies (i.e. values associated to frequencies less than a threshold frequency) in both relations $R_i$ and $S_i$. Tuples having these values for the join attribute have no effect neither on attribute value skew (AVS) nor on join product skew. These tuples will be redistributed using a hashing function.
- values $v \in \widetilde{Hist}_j^{a_i}(R_i \bowtie S_i)$ are associated to high frequencies. The communication templates are performed to avoid the effect of attribute values and join product skews of these values.

The communication templates are performed jointly by all processors (cf. [4,5,6]), **each one not necessarily computing the list of its own messages, so as to balance the overall process**. This step is computed (cf. [5,6]) in time:

$$Time_{c.2} \quad = \quad O\Big(g * |\widetilde{Hist}^{a_i}(R_i \bowtie S_i)| + \|\widetilde{Hist}^{a_i}(R_i \bowtie S_i)\| + l\Big).$$

At the end of this step, each processor $j$ has a local knowledge on the manner of distributing the tuples which will present in the join result, and the redistribution is done in time (cf. [5,6]):

$$Time_{c.3} \quad = \quad O\Big(c_{i/o} * \max_{j=1,\ldots,p}(|R_{i,j}| + |S_{i,j}|) + g * (|R_i \ltimes S_i| + |S_i \ltimes R_i|) + l\Big).$$

The term $c_{i/o} * (|R_{i,j}| + |S_{i,j}|)$ is the cost of a sequential traversal of relations $R_{i,j}$ and $S_{i,j}$ on processor $j$. We recall that, we redistribute only tuples of relations $R_i \ltimes S_i$ and $S_i \ltimes R_i$. We also recall that $|R_i \ltimes S_i|$ (resp. $|S_i \ltimes R_i|$) is generally very small compared to the size $|R_i|$ (resp. $|S_i|$) of relation $R_i$. Hence, this phase costs at most:

$$\begin{aligned} Time_c &= Time_{c.1} + Time_{c.2} + Time_{c.3} \\ &\leq O\Big(c_{i/o} * \max_{j=1,\ldots,p}(|R_{i,j}| + |S_{i,j}|) + \max_{j=1,\ldots,p}(\|R_{i,j}| + |S_{i,j}\|) \\ &\qquad + g * (|R_i \ltimes S_i| + |S_i \ltimes R_i|) + l\Big) \end{aligned}$$

---

[†]Note that the size of $Hist^{a_i}(R_i \bowtie S_i)$ is generally very small compared to the sizes of the histograms $Hist^{a_i}(R_i)$ and $Hist^{a_i}(S_i)$ because $Hist^{a_i}(R_i \bowtie S_i)$ contains only values that appear in both relations $R_i$ and $S_i$.

We recall that, $\|Hist_j^{a_i}(R_i \bowtie S_i)\| \ll \|Hist_j^{a_i}(R_i)\| \ll \|R_{i,j}\|$.

**Phase d: Creating the global histograms $Hist^{a_{(i+1)}}(R_{(i+1)})$ of relation $R_{(i+1)}$ and local histograms $Hist^{a_{(i+1)}}(S_{(i+1)}$ of relation $S_{(i+1)}$.** Similarly to relation $R_1$, the global histogram fragments of relation $R_{(i+1)}$ on join attribute $a_{(i+1)}$ are computed in time ( cf. equation 4):

$$Time_{d.1} \quad = \quad O\Bigg( \min\Big( (g * |Hist^{a_{(i+1)}}(R_{(i+1)})| + \|Hist^{a_{(i+1)}}(R_{(i+1)})\|), \max_{j=1,\ldots,p} g * (|R_{(i+1),j}| + \|R_{(i+1),j}\|)\Big) + c_{i/o} * \max_{j=1,\ldots,p} |R_{(i+1),j}| + l\Bigg)$$

Local histograms $Hist^{a_{(i+1)}}(S_{(i+1),j})$ of relation $S_{(i+1)} = R_i \bowtie S_i$ on attribute $a_{(i+1)}$ are computed directly from the histograms $Hist^{a_i}(R_i)$ and $Hist^{a_i}(S_i)$ using the following algorithm:

**Algorithm 2** Join result histogram creation algorithm

**Par** (on each node) $j \in [1,p]$ **do**
 ▷ $Hist^{a_{(i+1)}}(S_{(i+1),j})$=NULL (Create a B-tree to store histogram entries)
 **For each tuple $t$ of relation $R_{i,j}$ do**
  ▷ $freq1 = Hist^{a_i}(S_i)(t.a_i)$ (i.e. the frequency of the value of the attribute $a_i$ of $t$)
  **If** $(freq1 > 0)$ (i.e. tuple $t$ will be present in $R_i \bowtie S_i$) **Then**
   ▷ $freq2 = Hist^{a_{(i+1)}}(S_{(i+1),j})(t.a_{(i+1)})$
   **If** $(freq2 > 0)$ (i.e value $t.a_{(i+1)}$ is present in $Hist^{a_{(i+1)}}(S_{(i+1),j})$) **Then**
    ▷ Update $Hist^{a_{(i+1)}}(S_{(i+1)})(t.a_{(i+1),j}) = freq1 + freq2$
   **Else**
    ▷ Insert a new couple $(t.a_{(i+1)}, freq1)$ into the histogram $Hist^{a_{(i+1)}}(S_{(i+1),j})$
   **EndIf**
  **EndIf**
 **EndFor**
**EndPar**

Owing to the fact that the access to the histogram (equivalent to a search in a B-tree) is performed in a constant time, the cost of this step is:

$$Time_{d.2} \quad = \quad O\Big( \max_{j=1,\ldots,p} (\|R_{i,j}\|\Big)$$

On each processor $j$, the histogram $Hist^{a_{(i+1)}}(S_{(i+1),j})$ is then updated after the creation of communication templates by decreasing the frequency of a value $v$ whenever a tuple associated to value $v$ is sent from processor $j$ to another processor and, increasing the frequency of a value $v$ whenever a tuple associated to value $v$ is received by processor $j$. The communication templates are computed only for tuples that will be present in the join result. The cost of this step is:

$$Time_{d.3} \quad = \quad O\Big( \max_{j=1,\ldots,p} (\|R_{i,j} \ltimes S_i\|\Big)$$

We recall that $\|R_{i,j} \ltimes S_i\|$ is generally much smaller than $\|R_{i,j}\|$. The global cost of this phase is thus:

$$Time_d \quad = \quad Time_{d.1} + Time_{d.2} + Time_{d.3}$$

**Phase e: Computing the join result of $R_i$ and $S_i$.** Similarly to the algorithms `sfa_join` and `Osfa_join`, local joins of $R_{i,j}$ and $S_{i,j}$ are computed in time:

$$\begin{aligned} Time_e &= O\Big(c_{i/o} * \max_{i=1,\ldots,p}(|\widetilde{R'_{i,j}} \bowtie \widetilde{S''}_i| + |\widetilde{R''}_i \bowtie \widetilde{S'}_{i,j}| + |\widetilde{R'''}_{i,j} \bowtie \widetilde{S'''}_{i,j}|) \\ &\qquad + g * (|\widetilde{R''}_i| + |\widetilde{S''}_i|) + l\Big) \end{aligned}$$

where $\widetilde{R''_i}$ (resp. $\widetilde{S''_i}$) is the sub-relation of relation $\|R_i \ltimes S_i\|$ (resp. $\|S_i \ltimes R_i\|$) associated to high frequencies for the join attribute (see [5,6] for more details). The global cost of the pipelined strategy of `Osfa_join` for a n-join query is:

$$Time_{global\_query} = \sum_{i=1}^{i=n} Join\_time(i)$$

where $Join\_time(i)$ is time to compute join $i$ and

$$Join\_time(i) = \begin{cases} Time_a + Time_c + Time_d + Time_e & if \quad i = 1 \\ Time_b + Time_c + Time_d + Time_e & if \quad i > 1 \quad and \quad i < n \\ Time_b + Time_c + Time_e & if \quad i = n \end{cases}$$

All the terms of $Time_{global\_query}$ are bounded by those of a sequential parallel execution of `Osfa_join`. The main difference lies in costs of the histograms for intermediate join results: in the pipelined `Osfa_join` these histograms are computed from communication templates and the histograms of base relations (this step does not require to read data from disks) while histograms in a sequential parallel execution, are created from data stored on disk. We mention that pipelined `Osfa_join` offers flexible strategies for the resource allocation while avoiding unnecessary disk input/output.

### *4.3. Example*

Figures 1 and 2 represent an annotated query execution tree and illustrate the whole mechanism of the pipelined `Osfa_join`. This query tree corresponds to a 3-join query $R1 \bowtie S1 \bowtie R2 \bowtie R3$. Source tables are placed on leaves of the query tree. The number indicates the relative order of each operation.

Figure 1 illustrates an execution where operators are executed sequentially (i.e. a basic use of `Osfa_join`, where we do not use inter-operator parallelism). We first build the histograms for R1 and S1, then compute the communication templates, distribute data, and execute the join. Once the join is over, we build the histograms for $R2$ and $S2 = R1 \bowtie S1$, and so on.

Figure 2 illustrates a pipelined execution of the query tree. Building the histograms of all source tables is executed in parallel. As soon as histograms are available for $R1$ and $S1$, the communication template of the first join is computed, then data is transfered, the histogram for the next join is prepared and finally the join is executed. Once this phase is over, we can compute the communication template for the second join, and so on.

## 5. Conclusion

In this paper we proposed to adapt `Osfa_join` to pipelining throughout the use of a multi-attribute pre-scan strategy. This strategy consists in evaluating the

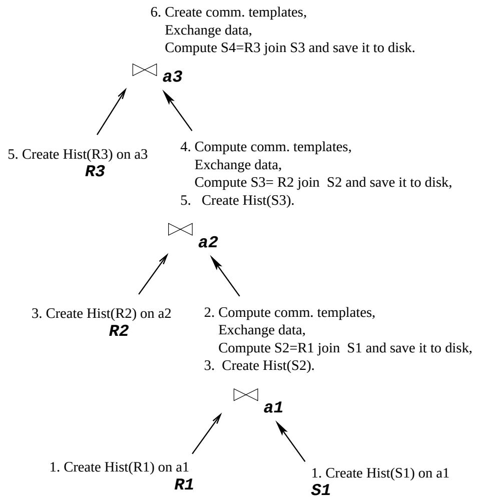


Figure 1: Standard execution of a multi-join query using `Osfa_join`

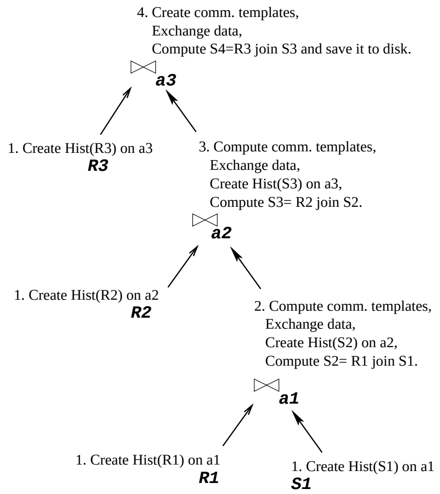


Figure 2: Pipelined execution of a multi-join query using `Osfa_join`

correlation between the join attributes of two successive joins. As a consequence, we can forecast the attribute frequencies of intermediate results, and then the way they should be redistributed for the next join, before their actual computation occurs. Pipeline is then possible and suitable. We used the BSP cost model to show that the pipelined version of `Osfa_join` offers a flexible strategy for resource allocation and reduces the cost of disk input/output for intermediate results.

The pipelined version of `Osfa_join` is currently the subject of a development in a parallel database prototype within the framework of the CARAML Grid project (http://www.caraml.org).

Future work will consist in studying how the multi-attribute pre-scan strategy could be used within longer pipeline chains. We will need to evaluate whether pre-scan can be applied to forecast the result of several consecutive joins, as its storage and computing costs might dramatically grow with the number of joins. Besides, we must notice that the length of pipeline chains is limited by the availability of nodes. While pipeline chains can be controlled by the use of segmented execution trees, this notion does not really apply to `Osfa_join`, which is a symmetric algorithm that does not distinguish between *built* and *probe* tables. We thus need to adapt the segmentation of execution trees to the semantics of our approach. We also plan to study the way these forecasts could help for resource allocation.

## References

[1] 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.

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

[3] 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.

[4] 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. Appears also in Progress in Computer Research, F. Columbus Ed. Vol. II, Nova Science Publishers, 2001.

[5] 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*, volume 1873 of *Lecture Notes in Computer Science*, pages 644–653, London, United Kingdom, 2000. Springer-Verlag.

[6] M. Bamha. An optimal and skew-insensitive join algorithm for shared nothing machines. Research Report RR-LIFO-2000-05, LIFO, Université d'Orléans, France, 2000.

[7] 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.

[8] 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.

[9] V. Poosala and Y. E. Ioannidis. Estimation of query-result distribution and its application in parallel-join load balancing. In *Proc. of the 22th International Conference on Very Large Database Systems*, pages 448–459, Bombay, India, September 1996.

[10] A. N. Wilschut, J. Flokstra, and P. M.G. Apers. Parallel evaluation of multi-join queries. In *Proc. of the ACM-SIGMOD*, volume 24, pages 115–126, San-Jose, California, USA, 1995.

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

[12] 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.

[13] 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, *Proc. of the ACM SIGMOD, Portland, Oregon*, pages 110–121, New York, NY 10036, USA, 1989. ACM Press.

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

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

[16] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[17] M.-S. Chen, P. S. Yu, and K.-L. Wu. Scheduling and processor allocation for the execution of multi-join queries. In *International Conference on Data Engineering*, pages 58–67, Los Alamos, Ca., USA, 1992. IEEE Computer Society Press.

[18] Erhard Rahm. Dynamic load balancing in parallel database systems. In *Proc. EURO-PAR'96 Conference, LNCS, Springer-Verlag, Lyon*, August 1996.

[19] M-S. Chen, M. L. Lo, P. S. Yu, and H. C. Young. Using segmented right-deep trees for the execution of pipelined hash joins. In Li-Yan Yuan, editor, *Proc. of the 18th International Conference on Very Large Data Bases*, pages 15–26, Vancouver, Canada, August 1992. Morgan Kaufmann Publishers.