AN OPTIMAL EVALUATION OF ”GROUPBY-JOIN” QUERIES IN DISTRIBUTED ARCHITECTURES

M. Al Hajj Hassan and M. Bamha
LIFO, Université d'Orléans, B.P. 6759, 45067 Orléans Cedex 2, France
{*mohamad.alhajjhassan,mostafa.bamha*}*@univ-orleans.fr*

Keywords: PDBMS[a], Parallel joins, Data skew, Join product skew, GroupBy-Join queries, BSP cost model.

[a]PDBMS : Parallel Database Management Systems.

Abstract: SQL queries involving join and group-by operations are fairly common in many decision support applications where the size of the input relations is usually very large, so the parallelization of these queries is highly recommended in order to obtain a desirable response time. Several parallel algorithms that treat this kind of queries have been presented in the literature. However, their most significant drawbacks are that they are very sensitive to data skew and involve expansive communication and Input/Output costs in the evaluation of the join operation. In this paper, we present an algorithm that overcomes these drawbacks because it evaluates the ”GroupBy-Join” query without the need of the direct evaluation of the costly join operation, thus reducing its Input/Output and communication costs. Furthermore, the performance of this algorithm is analyzed using the scalable and portable BSP (Bulk Synchronous Parallel) cost model which predicts a linear speedup even for highly skewed data.

1 Introduction

Aggregate functions used to summarize large volume of data based on a designated grouping are widely employed in applications such as: the decision support application, OnLine Analytical Processing (OLAP) and Data Warehouse (Gupta et al., 1995; Li et al., 2005; Taniar and Rahayu, 2001), because in such applications, aggregated and summarized data are more important than detailed records (Datta et al., 1998). Aggregate operations may be applied on the output of the join operation of multiple tables having potentially billions of records. These tables may rapidly grow every day especially in OLAP systems (Datta et al., 1998). Moreover, the output of these queries must be obtained in a reasonable processing time. For these reasons, parallel processing of queries involving group-by and join operations results in huge performance gain, especially in the presence of parallel DBMS (PDBMS). However, the use of efficient parallel algorithm in PDBMS is fundamental in order to obtain an acceptable performance (Bamha and Hains, 2000; Bamha and Hains, 1999; Mourad et al., 1994; Seetha and Yu, 1990).

Several parallel algorithms for evaluating ”GroupBy-Join” queries were presented in the literature (Shatdal and Naughton, 1995; Taniar et al., 2000), but these algorithms are inefficient due to the following reasons:

1. The communication cost in these algorithms is very high because all the tuples of the relations are redistributed between processors. Some of these tuples may not even contribute in the result of the join operation.
2. These algorithms fully materialize the intermediate results of the join operations. This is a significant drawback because the size of the result of this operation is generally large with respect to the size of the input relations. In addition, the Input/Output cost in these algorithms is very high where it is reasonable to assume that the output relation cannot fit in the main memory of every processor, so it must be reread from disk in order

to evaluate the aggregate function.

3. These algorithms cannot solve the problem of data skew because data redistribution is generally based on hashing data into buckets and hashing is known to be inefficient in the presence of high frequencies (Bamha, 2005; Schneider and DeWitt, 1989; Seetha and Yu, 1990).

In this paper, we present a new parallel algorithm used to evaluate the ”GroupBy-Join” queries on Shared Nothing machines (a distributed architecture where each processor has its own memory and own disks), when the join attributes are different from the group-by attributes. Our main contribution is that, in this algorithm, we do not need to materialize the join operation as in the traditional algorithms where the join operation is evaluated first and then the group-by and aggregate functions (Yan and Larson, 1994). This algorithm is also insensitive to data skew and its communication and Input/Output costs are reduced to a minimum.

In this algorithm, we partially evaluate the aggregate function before redistributing the tuples. This helps in reducing the cost of data redistribution. We use the histograms of both relations in order to find the tuples that participate in the result of the join operation. It is proved in (Bamha and Hains, 2005; Bamha and Hains, 1999), using the BSP model, that the histogram management has a negligible cost when compared to the gain it provides in reducing the communication cost.
In traditional algorithms, all the tuples of the output of the join operation are redistributed using a hashing function. In the contrary, in our algorithm we only redistribute the result of the semi-join of the histograms which are very small compared to the size of input relations. The use of semi-join in multi-processor machines to evaluate the ”GroupBy-Join” queries helps in reducing the amount of data transferred over the network and therefore the communication cost in distributed systems (Chen and Yu, 1993; Stocker et al., 2001). The performance of this algorithm is analyzed using the BSP cost model (Skillicorn et al., 1997) which predicts for our algorithm a linear speedup even for highly skewed data.

The rest of the paper is organized as follows. In section 2, we present the BSP cost model used to evaluate the processing time of different phases of the algorithm. In section 3, we give an overview of different computation methods of ”GroupBy-Join” queries. In section 4, we describe our algorithm. We then conclude in section 5.

2 The BSP Cost Model

Bulk-Synchronous Parallel (BSP) cost model is a programming model introduced by L. Valiant (Valiant, 1990) to offer a high degree of abstraction like PRAM models and yet allow portable and predictable performance on a wide variety of multi-processor architectures (Skillicorn et al., 1997). 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,
- 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.

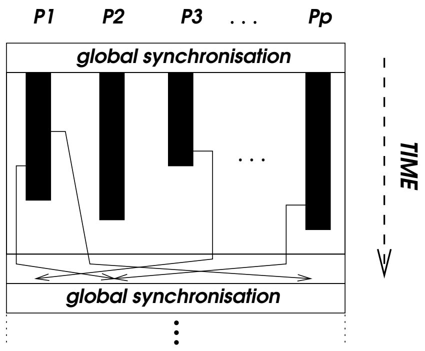

Figure 1: A BSP superstep.

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 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 $w_i^{(s)}$ is the 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).

3 "GroupBy-Join" Queries Computation

In DBMS, the aggregate functions can be applied on the tuples of a single table, but in most SQL queries, they are applied on the output of the join of multiple relations. In the later case, we can distinguish two types of "GroupBy-Join" queries. We will illustrate these two types using the following example.
In this example, we have three relations that represent respectively Suppliers, Products and quantity of a product shipped by a supplier in a specific date.

```
SUPPLIER (Sid, Sname, City)
PRODUCT (Pid, Pname, Category)
SHIPMENT (Sid, Pid, Date, Quantity)
```

Query 1

```
Select p.Pid, p.Pname, SUM (Quantity)
From PRODUCT as p, SHIPMENT as s
Where p.Pid = s.Pid
Group By p.Pid
```

Query 2

```
Select p.Category, SUM (Quantity)
From PRODUCT as p, SHIPMENT as s
Where p.Pid = s.Pid
Group By p.Category
```

The purpose of $Query1$ is to find the total quantity of every product shipped by all the suppliers, while that of $Query2$ is to find the total amount of every category of product shipped by all the suppliers.
The difference between $Query1$ and $Query2$ lies in the group-by and join attributes. In $Query1$, the join attribute (Pid) and the group-by attribute are the same. In this case, it is preferable to carry out the group-by and aggregate functions first and then the join operation (Taniar et al., 2000; Taniar and Rahayu, 2001), because this helps in reducing the size of the relations to be joined. As a consequence, applying the group-by and aggregate functions before the join operation in PDBMS results in a huge gain in the communication cost and the execution time of the "GroupBy-Join" queries.
In the contrary, this can not be applied on Query 2, because the join attribute (Pid) is different from the group-by attribute ($category$). In this paper, we will focus on this type of "GroupBy-Join" queries. In the traditional algorithms that treat this kind of queries, the costly join operation is evaluated in the first step and then the aggregate function (Taniar et al., 2000; Taniar and Rahayu, 2001). However, in our algorithm, we succeeded to partially evaluate the aggregate functions before redistributing the tuples using histograms, thus reducing the communication cost as much as possible.

4 GroupBy-Join Queries: A new approach

In this section, we present a detailed description of a new parallel algorithm used to evaluate the "GroupBy-Join" queries when the group-by attributes are different from the join attributes. We assume that the 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.
For simplicity of description and without loss of generality, we consider that the query has only one join attribute x and that the group-by attribute set consists of one attribute y of R and another attribute z of S. We also assume that the aggregate function is applied on the values of the attribute u of S.

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, a sub-relation of T,
- $Hist^w(T)$ denotes the histogram[1] of relation T with respect to the attribute w, 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 attribute w. The histogram is often much smaller and never larger than the relation it describes,
- $Hist^w(T_i)$ denotes the histogram of fragment T_i,

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

- $Hist_i^w(T)$ is processor i's fragment of the histogram of T,
- $Hist^w(T)(v)$ is the frequency (n_v) of value v in relation T,
- $Hist^w(T_i)(v)$ is the frequency of value v in sub-relation T_i,
- $AGGR_{f,u}^w(T)$ [2] is the result of applying the aggregate function f on the values of the attribute u of every group of tuples of T having identical values of the group-by attributes w. $AGGR_{f,u}^w(T)$ is formed of a list of tuples (v, f_v) where f_v is the result of the aggregate function of the group of tuples having value v for the attribute w (w may be formed of more than one attribute),
- $AGGR_{f,u}^w(T_i)$ denotes the result of applying the aggregate function on the attribute u of relation T_i,
- $AGGR_{f,u,i}^w(T)$ is processor i's fragment of the result of applying the aggregate function on T,
- $AGGR_{f,u}^w(T)(v)$ is the result f_v of the aggregate function of the group of tuples having value v for the group-by attribute w in relation T,
- $AGGR_{f,u}^w(T_i)(v)$ is the result f_v of the aggregate function of the group of tuples having value v for the group-by attribute w in sub-relation T_i,
- $\|T\|$ denotes the number of tuples of relation T, and
- $|T|$ denotes the size (expressed in bytes or number of pages) of relation T.

The algorithm proceeds in six phases. We will give an upper bound of the execution time of each superstep using BSP cost model. The notation $O(\ldots)$ hides only small constant factors: they depend only on the program implementation but neither on data nor on the BSP machine parameters.

Phase 1: Creating local histograms

In this phase, the local histograms $Hist^{x,y}(R_i)(i = 1,\ldots,p)$ of blocks R_i are created in parallel by a scan of the fragment R_i on processor i in time $c_{i/o} * \max_{i=1,\ldots,p} |R_i|$ where $c_{i/o}$ is the cost of writing/reading a page of data from disk.

In addition, the local fragments $AGGR_{f,u}^{x,z}(S_i)(i = 1,\ldots,p)$ of blocks S_i are also created in parallel on each processor i by applying the aggregate function f on every group of tuples having identical values of the couple of attributes (x,z) in time $c_{i/o} * \max_{i=1,\ldots,p} |S_i|$.

[2] $AGGR_{f,u}^w(T)$ is implemented as a balanced tree (B-tree).

(In this algorithm the aggregate function may be MAX, MIN, SUM or $COUNT$. For the aggregate AVG a similar algorithm that merges the $COUNT$ and the SUM algorithms is applied).

In this algorithm, we only redistribute the tuples of $Hist^{x,y}(R_i)$ and $AGGR_{f,u}^{x,z}(S_i)$ that participate effectively in the join result. These tuples are determined in phase 2, but we need first to compute the frequency of each value of the attribute x in $Hist^{x,y}(R_i)$ and $AGGR_{f,u}^{x,z}(S_i)$. So while creating $Hist^{x,y}(R_i)$ (resp. $AGGR_{f,u}^{x,z}(S_i)$), we also create on the fly their local histograms $Hist'^x(R_i)$ (resp. $Hist'^x(S_i)$) with respect to x, i.e. $Hist'^x(R_i)$ (resp. $Hist'^x(S_i)$) holds the frequency of each value of the attribute x in $Hist^{x,y}(R_i)$ (resp. $AGGR_{f,u}^{x,z}(S_i)$) for $i = 1,\ldots,p$.
In fact, the difference between $Hist^x(R_i)$ and $Hist'^x(R_i)$ is that $Hist^x(R_i)$ holds the frequency of each value of the attribute x in relation R_i (i.e., for each value d of the attribute x, we find the number of tuples of R_i having the value d of x), while in $Hist'^x(R_i)$ we count tuples having the same values of the attributes (x,y) only once.
We use the following algorithm to create $Hist'^x(R_i)$ and a similar one is used to create $Hist'^x(S_i)$.

```
Par (on each node in parallel) $i = 1,\ldots,p$
  $Hist'^x(R_i)$ = NULL [3]
  For every tuple $t$ that will be inserted or used to
   modify $Hist^{x,y}(R_i)$ do
    If $Hist^{x,y}(R_i)(t.x,t.y)$ = NULL Then [4]
      $freq_1$ = $Hist'^x(R_i)(t.x)$
      If $freq_1 \neq$ NULL Then
        Increment the frequency of $t.x$ in $Hist'^x(R_i)$
      Else
        Insert a new tuple $(t.x,1)$ into $Hist'^x(R_i)$
      EndIf
    EndIf
  EndFor
EndPar
```

In principle, this phase costs:

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

Phase 2: Local semi-joins computation

In order to minimize the communication cost, only tuples of $Hist^{x,y}(R)$ and $AGGR_{f,u}^{x,z}(S)$ that will be present in the join result are redistributed. To this end, we compute the following local semi-joins: $\overline{Hist}^{x,y}(R_i) = Hist^{x,y}(R_i) \ltimes AGGR_{f,u}^{x,z}(S)$ and $\overline{AGGR}_{f,u}^{x,z}(S_i) = AGGR_{f,u}^{x,z}(S_i) \ltimes Hist^{x,y}(R)$. To compute these semi-joins, we use proposition 2 presented in (Bamha and Hains, 2005), but instead of applying the hashing function on the tuples of $Hist^x(R_i)$ and $Hist^x(S_i)$ to compute the global histograms, we apply it here on the tuples of $Hist'^x(R_i)$ and $Hist'^x(S_i)$. In fact the number of tuples of $Hist'^x(R_i)$ and that of $Hist^x(R_i)$

are equal, what differs is only the value of the frequency attribute in these histograms, so $|Hist'^{x}(R_i)| = |Hist^{x}(R_i)|$ (this also applies to $Hist'^{x}(S_i)$ and $Hist^{x}(S_i)$). Hence the cost of this phase is (Bamha and Hains, 2005):

$$\begin{aligned}
Time_{phase2} &= \\
O\Big(&\max_{i=1,\dots,p} ||Hist^{x,y}(R_i)|| + \max_{i=1,\dots,p} ||AGGR^{x,z}_{f,u}(S_i)|| + \\
&min\big(g * |Hist^{x}(R)| + ||Hist^{x}(R)||, g * \frac{|R|}{p} + \frac{||R||}{p}\big) + \\
&min\big(g * |Hist^{x}(S)| + ||Hist^{x}(S)||, g * \frac{|S|}{p} + \frac{||S||}{p}\big) + l\Big),
\end{aligned}$$

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

We recall (cf. to proposition 1 in (Bamha and Hains, 2005)) that, in the above equation, the terms:

$$min\big(g * |Hist^{x}(R)| + ||Hist^{x}(R)||, g * \frac{|R|}{p} + \frac{||R||}{p}\big),$$

and

$$min\big(g * |Hist^{x}(S)| + ||Hist^{x}(S)||, g * \frac{|S|}{p} + \frac{||S||}{p}\big),$$

represent the necessary time to compute the global histograms $Hist'^{x}_{i=1,\dots,p}(R)$ and $Hist'^{x}_{i=1,\dots,p}(S)$, respectively starting from the local histograms $Hist'^{x}(R_i)$ and $Hist'^{x}(S_i)$ where $i = 1,\dots,p$.

During semi-join computation, we store an extra information called $index(d) \in \{1,2,3\}$ for each value $d \in Hist'^{x}(R) \cap Hist'^{x}(S)$. This information will allow us to decide if, for a given value d, the frequencies of tuples of $Hist^{x,y}(R)$ and $AGGR^{x,z}_{f,u}(S)$ 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 this algorithm, by evaluating $AGGR^{x,z}_{f,u}(S)$ we partially apply the aggregate function on the attribute u of S thus reducing the volume of data, this also applies to $Hist^{x,y}(R)$ where all tuples having the same values of (x,y) are represented by a single tuple, but we will still consider that the frequencies of some tuples of $AGGR^{x,z}_{f,u}(S)$ and $Hist^{x,y}(R)$ having a value d of the attribute x is high. So in order to balance the load of all the processors, these tuples must be evenly redistributed.

In the rest of this paper, we use the same threshold frequency as in fa-join algorithm (Bamha and Hains, 2000; Bamha and Hains, 1999), i.e. $f_0 = p * log(p)$.

For a given value $d \in Hist'^{x}(R) \cap Hist'^{x}(S)$ [5],

[5]The intersection of $Hist'^{x}(R)$ and $Hist'^{x}(S)$ is found while computing the semi-joins (c.f proposition 2 presented in (Bamha and Hains, 2005))

- the value $index(d) = 3$, means that the frequency of tuples of relations $Hist^{x,y}(R)$ and $AGGR^{x,z}_{f,u}(S)$ associated to value d are less than the threshold frequency. (i.e. $Hist'^{x}(R)(d) < f_0$ and $Hist'^{x}(S)(d) < f_0$),
- the value $index(d) = 2$, means that $Hist'^{x}(S)(d) \geq f_0$ and $Hist' x(S)(d) > Hist'^{x}(R)(d)$,
- the value $index(d) = 1$, means that $Hist'^{x}(R)(d) \geq f_0$ and $Hist'^{x}(R)(d) \geq Hist'^{x}(S)(d)$.

Note that unlike the algorithms presented in (Shatdal and Naughton, 1995; Taniar et al., 2000) where both relations R and S are redistributed, we will only redistribute $Hist^{x,y}(R_i) \ltimes AGGR^{x,z}_{f,u}(S)$ and $AGGR^{x,z}_{f,u}(S_i) \ltimes Hist^{x,y}(R)$ to find the final result. This will reduce the communication costs to a minimum.

At the end of this phase, we will divide the semi-joins $\overline{Hist}^{x,y}(R_i)$ and $\overline{AGGR}^{x,z}_{f,u}(S_i)$ on each processor i into three sub-histograms in the following way:

$$\overline{Hist}^{x,y}(R_i) = \bigcup_{j=1}^{3} \overline{Hist}^{(j)x,y}(R_i)$$

and

$$\overline{AGGR}^{x,z}_{f,u}(S_i) = \bigcup_{j=1}^{3} \overline{AGGR}^{(j)x,z}_{f,u}(S_i)$$

where:

- All the tuples of $\overline{Hist}^{(1)x,y}(R_i)$ (resp. $\overline{AGGR}^{(1)x,z}_{f,u}(S_i)$) are associated to values d such that $index(d) = 1$ (resp. $index(d) = 2$),
- All the tuples of $\overline{Hist}^{(2)x,y}(R_i)$ (resp. $\overline{AGGR}^{(2)x,z}_{f,u}(S_i)$) are associated to values d such that $index(d) = 2$ (resp. $index(d) = 1$),
- All the tuples of $\overline{Hist}^{(3)x,y}(R_i)$ and $\overline{AGGR}^{(3)x,z}_{f,u}(S_i)$ 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.

The tuples of $\overline{Hist}^{(1)x,y}(R_i)$ and $\overline{AGGR}^{(1)x,z}_{f,u}(S_i)$ are associated to high frequencies for the join attribute. These tuples have an important effect on Attribute Value Skew (AVS) and Join Product Skew (JPS). So we will use an appropriate redistribution algorithm in order to efficiently avoid both AVS and JPS. However the tuples of relations $\overline{Hist}^{(3)x,y}(R_i)$ and $\overline{AGGR}^{(3)x,z}_{f,u}(S_i)$ (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: Creating the 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 join algorithms. To avoid the slowdown usually caused by AVS and the imbalance of the size of local joins processed by the standard join algorithms, an appropriate treatment for high attribute frequencies is needed (Bamha and Hains, 1999; Bamha and Hains, 2000; Bamha, 2005).

3.a To this end, we partition the histogram $Hist'^{x}(R \bowtie S)$ [6] into two sub-histograms: $Hist^{(1,2)'x}(R \bowtie S)$ and $Hist^{(3)'x}(R \bowtie S)$ in the following manner:

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

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

3.b Communication templates for high frequencies:
We first create a communication template: the list of messages which constitutes 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.
So each processor i computes a set of necessary messages relating to the values d it owns in $Hist_i^{(1,2)'x}(R \bowtie S)$. The communication template is derived by applying the following algorithm on the tuples of relations $\overline{Hist}^{(1)x,y}(R)$ which is mapped to multiple nodes. We also apply the same algorithm to compute the communication template of $\overline{AGGR}_{f,u}^{(1)x,z}(S)$, but we replace $Hist'^{x}(R)$ by $Hist'^{x}(S)$.

[6] $Hist'^{x}(R \bowtie S)$ is simply the intersection of $Hist'^{x}(R)$ and $Hist'^{x}(S)$.

```
if $\big(Hist'^{x}(R)(d) mod(p) = 0\big)$ then
  each processor $j$ will hold a block of size
  $block_j(d) = \frac{Hist'^{x}(R)(d)}{p}$ of tuples of value $d$.
else
  begin
    Pick a random value $j_0$ between 0 and $(p-1)$
    if (processor's index $j$ is between $j_0$ and
        $j_0 + \big(Hist'^{x}(R)(d)\ mod\ p\big)$) then
      the processor of index $j$ will hold a block
      of size: $block_j(d) = \lfloor \frac{Hist'^{x}(R)(d)}{p} \rfloor + 1$
    else
      the processor of index $j$ will hold a block
      of size: $block_j(d) = \lfloor \frac{Hist'^{x}(R)(d)}{p} \rfloor$
end.
```

In the above algorithm, $\lfloor x \rfloor$ is the largest integral value not greater than x and $block_j(d)$ is 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 $d \in Hist_i^{(1,2)'x}(R \bowtie S)$, 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. $Rest_j(d) < 0$) send (resp. receive) tuples of value of d. This step is thus completed in time: $Time_{3.b} = O\Big(||Hist^{(1,2)'x}(R \bowtie S)||\Big)$.

The tuples associated to low frequencies (i.e. tuples having $d \in Hist_i^{(3)'x}(R \bowtie S)$) 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 the communication templates has therefore taken the sum of the above two steps:

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

Phase 4: Data redistribution

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

Every processor i holds, for every one of its local $d \in Hist_i^{(1,2)'x}(R \bowtie S)$, 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 processor 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 into 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: $Rest_{proc(j)}(d) > 0$ *for* $j = 1, ..., \alpha$ and $Rest_{proc(j)}(d) < 0$ *for* $j = 1+\alpha, ..., \beta$
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 := α + 1;
while (i ≤ α) 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.
```

Figure 2 gives an example of the value *Rest* as-

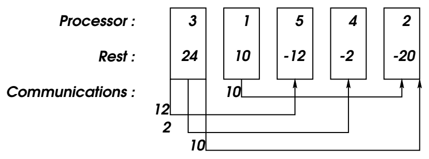

Figure 2: Sending orders as a function of *Rest* values.

sociated to a value of the join attribute and the corresponding sending orders.

The maximal complexity of this algorithm is: $O\Big(||Hist^{(1,2)'x}(R \bowtie S)||\Big)$ 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 $d \in Hist_i^{(1,2)'x}(R \bowtie S)$.
For each processor i and $d \in Hist_i^{(1,2)'x}(R \bowtie S)$, all the order_to_send$(j, i, ...)$ are sent to processor j when $j \neq i$ in time $O\big(g * |Hist^{(1,2)'x}(R \bowtie S)| + l\big)$.
Thus, this step costs:

$$Time_{4.a} = O\big(g * |Hist^{(1,2)'x}(R \bowtie S)| + ||Hist^{(1,2)'x}(R \bowtie S)|| + l\big).$$

4.b Redistribution of tuples with values $d \in Hist_i^{(3)'x}(R \bowtie S)$:
Tuples of $\overline{Hist}^{(3)x,y}(R_i)$ and $\overline{AGGR}^{(3)x,z}_{f,u}(S_i)$ (i.e. tuples having $d \in Hist_i^{(3)'x}(R \bowtie S)$) 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 $\overline{Hist}^{x,y}(R_i)$ and $\overline{AGGR}^{x,z}_{f,u}(S_i)$ will be redistributed. Redistribution is then performed in time:
$Time_{4.b} = O\Big(g * \big(|\overline{Hist}^{x,y}(R_i)| + |\overline{AGGR}^{x,z}_{f,u}(S_i)|\big) + l\Big)$.
Thus the total cost of the redistribution phase is the sum of the costs of the above two steps:

$$Time_{phase4} = O\Big(g * \max_{i=1,...,p} \big(|\overline{Hist}^{x,y}(R_i)| + |\overline{AGGR}^{x,z}_{f,u}(S_i)| + |Hist^{(1,2)'x}(R \bowtie S)|\big) + ||Hist^{(1,2)'x}(R \bowtie S)|| + l\Big)$$

We mention that we only redistribute the tuples of the semi-joins $\overline{Hist}^{x,y}(R_i)$ and $\overline{AGGR}^{x,z}_{f,u}(S_i)$ where $|\overline{Hist}^{x,y}(R_i)|$ and $|\overline{AGGR}^{x,z}_{f,u}(S_i)|$ are generally very small compared to $|R_i|$ and $|S_i|$. In addition $|Hist'^{x}(R \bowtie S)|$ is generally very small compared to $|Hist^{x,y}(R)|$ and $|AGGR^{x,z}_{f,u}(S)|$. Thus we reduce the communication cost to a minimum.

Phase 5: local computation of the aggregate function

At this step, every processor has partitions of $\overline{Hist}^{x,y}(R)$ and $\overline{AGGR}^{x,z}_{f,u}(S)$. Using equation 2 in (Bamha, 2005), we can deduce that the tuples of $\overline{Hist}^{(1)x,y}(R_i)$, $\overline{Hist}^{(2)x,y}(R_i)$, $\overline{Hist}^{(3)x,y}(R_i)$ can be joined with the tuples of $\overline{AGGR}^{(2)x,z}_{f,u}(S_i)$, $\overline{AGGR}^{(1)x,z}_{f,u}(S_i)$, $\overline{AGGR}^{(3)x,z}_{f,u}(S_i)$ respectively. But the frequencies of tuples of $\overline{Hist}^{(1)x,y}(R_i)$ and $\overline{AGGR}^{(1)x,z}_{f,u}(S_i)$ are by definition greater than the corresponding (matching) tuples in $\overline{Hist}^{(2)x,y}(R_i)$ and $\overline{AGGR}^{(2)x,z}_{f,u}(S_i)$ respectively. So we will choose $\overline{Hist}^{(1)x,y}(R_i)$ and $\overline{AGGR}^{(1)x,z}_{f,u}(S_i)$ as the *build* relations and $\overline{Hist}^{(2)x,y}(R_i)$ and $\overline{AGGR}^{(2)x,z}_{f,u}(S_i)$ as *probe* relations. Hence, we need to duplicate the probe relations to all processors in time:

$$Time_{phase5.a} = O\Big(g * \big(|\overline{Hist}^{(2)x,y}(R)| + |\overline{AGGR}^{(2)x,z}_{f,u}(S)|\big) + l\Big).$$

Now, using the following Algorithm, we are able to compute the local aggregate function on every processor without the necessity to fully materialize the intermediate results of the join operation.

In this algorithm, we create on each processor i, the relation $AGGR^{y,z}_{f,u}((R \ltimes S)_i)$ that holds the local results of applying the aggregate function on every group of tuples having the value of the couple of attributes (y,z). $AGGR^{y,z}_{f,u}((R \ltimes S)_i)$ has the form (y,z,v) where y and z are the group-by attributes and v is the result of the aggregate function.

```
Par (on each node in parallel) $i = 1,...,p$
  $AGGR^{y,z}_{f,u}((R \ltimes S)_i)$ = NULL; [7]
  For every tuple $t$ of relation $\overline{Hist}^{(1)x,y}(R_i)$ do
    For every entry $v_1 = \overline{AGGR}^{(2)x,z}_{f,u}(S_i)(t.x,z)$ do
      $v_2 = AGGR^{y,z}_{f,u}((R \ltimes S)_i)(t.y,z)$;
      If $v_2 \neq$ NULL Then
        Update $AGGR^{y,z}_{f,u}((R \ltimes S)_i)(t.y,z)$ = $F(v_1,v_2)$
        where $F()$ is the aggregate function;
      Else
        Insert a new tuple $(t.y,z,v_1)$ into the
        histogram $AGGR^{y,z}_{f,u}((R \ltimes S)_i)$;
      EndIf
    EndFor
  EndFor
  For every tuple $t$ of relation $\overline{AGGR}^{(1)x,z}_{f,u}(S_i)(t.x,z)$ do
    For every entry $v_1 = \overline{Hist}^{(2)x,y}(R_i)$ do
      $v_2 = AGGR^{y,z}_{f,u}((R \ltimes S)_i)(t.y,z)$;
      If $v_2 \neq$ NULL Then
        Update $AGGR^{y,z}_{f,u}((R \ltimes S)_i)(t.y,z)$ = $F(v_1,v_2)$
        where $F()$ is the aggregate function;
      Else
        Insert a new tuple $(t.y,z,v_1)$ into the
        histogram $AGGR^{y,z}_{f,u}((R \ltimes S)_i)$;
      EndIf
    EndFor
  EndFor
  For every tuple $t$ of relation $\overline{Hist}^{(3)x,y}(R_i)$ do
    For every entry $v_1 = \overline{AGGR}^{(3)x,z}_{f,u}(S_i)(t.x,z)$ do
      $v_2 = AGGR^{y,z}_{f,u}((R \ltimes S)_i)(t.y,z)$;
      If $v_2 \neq$ NULL Then
        Update $AGGR^{y,z}_{f,u}((R \ltimes S)_i)(t.y,z)$ = $F(v_1,v_2)$
        where $F()$ is the aggregate function
      Else
        Insert a new tuple $(t.y,z,v_1)$ into the
        histogram $AGGR^{y,z}_{f,u}((R \ltimes S)_i)$
      EndIf
    EndFor
  EndFor
EndPar
```

The cost of applying this algorithm is:

$$Time_{phase5.b} = c_{i/o} * O\Big(\max_{i=1,...,p}\big(|\overline{Hist}^{(1)x,y}(R_i) \bowtie \overline{AGGR}^{(2)x,z}_{f,u}(S)| + |\overline{Hist}^{(2)x,y}(R) \bowtie \overline{AGGR}^{(1)x,z}_{f,u}(S_i)| + |\overline{Hist}^{(3)x,y}(R_i) \bowtie \overline{AGGR}^{(3)x,z}_{f,u}(S_i)|\big)\Big)$$

So the total cost of this phase is:

$$Time_{phase5} = O\Big(g * \big(|\overline{Hist}^{(2)x,y}(R)| + |\overline{AGGR}^{(2)x,z}_{f,u}(S)|\big) + c_{i/o} * \max_{i=1,...,p}\big(|\overline{Hist}^{(1)x,y}(R_i) \bowtie \overline{AGGR}^{(2)x,z}_{f,u}(S)| + |\overline{Hist}^{(2)x,y}(R) \bowtie \overline{AGGR}^{(1)x,z}_{f,u}(S_i)| + |\overline{Hist}^{(3)x,y}(R_i) \bowtie \overline{AGGR}^{(3)x,z}_{f,u}(S_i)|\big) + l\Big)$$

Phase 6: global computation of the aggregate function

In this phase, a global application of the aggregate function is carried out. For this purpose, every processor redistributes the local aggregation results, $AGGR^{y,z}_{f,u}((R \ltimes S)_i)$, using a common hashing function. The input attributes of the hashing function are y and z. After hashing, every processor applies the aggregate function on the received messages in order to compute the global result $AGGR^{y,z}_{f,u}(R \bowtie S)$.

$AGGR^{y,z}_{f,u}(R \bowtie S)$ is formed of three attributes. The first two are the group-by attributes (y and z) and the third is the result of the applying the aggregate function.
The time of this step is:

$$Time_{phase6} = O\Big(min\big(g * |AGGR^{y,z}_{f,u}(R \bowtie S)| + ||AGGR^{y,z}_{f,u}(R \bowtie S)||, g * \frac{|R \bowtie S|}{p} + \frac{||R \bowtie S||}{p}\big) + l\Big)$$

where we apply the same result used to redistribute the histograms (cf. to proposition 1 in (Bamha and Hains, 2005)) in redistributing $AGGR^{y,z}_{f,u}((R \ltimes S)_i)$.

Remark 1
In practice, the imbalance of the data related to the use of the hash functions can be due to :

- *a bad choice of the hash function* used. This imbalance can be avoided by using the hashing techniques presented in the literature making it possible to distribute evenly the values of the join attribute with a very high probability (Carter and Wegman, 1979),
- *an intrinsic data imbalance* which appears when some values of the join attribute appear more frequently than others. By definition a hash function maps tuples having the same join attribute values to the same processor. These is no way for a clever hash function to avoid load imbalance that result from these repeated values (DeWitt et al.,

1992). But this case cannot arise here owing to the fact that histograms contains only distinct values of the join attribute and the hashing functions we use are always applied to histograms.

The global cost of evaluating the "GroupBy-Join" queries in this algorithm is the sum of redistribution cost and local computation of aggregate function. It is of the order:

$$
\begin{aligned}
Time_{total} = O\Big(& c_{i/o} * \max_{i=1,\dots,p} (|R_i| + |S_i|) \\
& + min\big(g * |Hist^x(R)| + ||Hist^x(R)||, g * \frac{|R|}{p} + \frac{||R||}{p}\big) \\
& + min\big(g * |Hist^x(S)| + ||Hist^x(S)||, g * \frac{|S|}{p} + \frac{||S||}{p}\big) \\
& + g * \max_{i=1,\dots,p} \Big(|\overline{Hist}^{x,y}(R_i)| + |\overline{AGGR}^{x,z}_{f,u}(S_i)| \\
& \qquad + |Hist^{(1,2)'x}(R \bowtie S)|\Big) \\
& + ||Hist^{(1,2)'x}(R \bowtie S)|| \\
& + g * \big(|\overline{Hist}^{(2)x,y}(R)| + |\overline{AGGR}^{(2)x,z}_{f,u}(S)|\big) \\
& + c_{i/o} * \max_{i=1,\dots,p} \big(|\overline{Hist}^{(1)x,y}(R_i) \bowtie \overline{AGGR}^{(2)x,z}_{f,u}(S)| \\
& \qquad + |\overline{Hist}^{(2)x,y}(R) \bowtie \overline{AGGR}^{(1)x,z}_{f,u}(S_i)| \\
& \qquad + |\overline{Hist}^{(3)x,y}(R_i) \bowtie \overline{AGGR}^{(3)x,z}_{f,u}(S_i)|\big) \\
& + min\big(g * |AGGR^{y,z}_{f,u}(R \bowtie S)| + ||AGGR^{y,z}_{f,u}(R \bowtie S)||, \\
& \qquad g * \frac{|R \bowtie S|}{p} + \frac{||R \bowtie S||}{p}\big) \\
& + \max_{i=1,\dots,p} ||Hist^{x,y}(R_i)|| + \max_{i=1,\dots,p} ||AGGR^{x,z}_{f,u}(S_i)|| + l\Big).
\end{aligned}
$$

Remark 2
In the traditional algorithms, the aggregate function is applied on the output of the join operation. The sequential evaluation of the "groupBy-Join" queries requires at least the following lower bound:

$$bound_{inf_1} = \Omega\big(c_{i/o} * (|R| + |S| + |R \bowtie S|)\big).$$

Parallel processing with p processors requires therefore:

$$bound_{inf_p} = \frac{1}{p} * bound_{inf_1}.$$

Using our approach, the evaluation of the "GroupBy-Join" queries when the join attributes are different from the group-by attributes has an optimal asymptotic complexity when:

$$
\begin{aligned}
& max\big(|\overline{Hist}^{(2)x,y}(R)|, |\overline{AGGR}^{(2)x,z}_{f,u}(S)|, |Hist^{(1,2)'x}(R \bowtie S)|\big) \\
& \quad \leq c_{i/o} * max(\frac{|R|}{p}, \frac{|S|}{p}, \frac{|R \bowtie S|}{p}),
\end{aligned}
$$

this is due to the fact that the local join results have almost the same size and all the terms in $Time_{total}$ 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 presented a parallel algorithm used to compute "GroupBy-Join" queries in a distributed architecture when the group-by attributes and the join attributes are not the same. This algorithm can be used efficiently to reduce the execution time of the query, because we do not materialize the costly join operation which is a necessary step in all the other algorithms presented in the literature that treat this type of queries, thus reducing the Input/Output cost. It also helps us to balance the load of all the processors even in the presence of AVS and to avoid the JPS which may result from computing the intermediate join results.

In addition, the communication cost is reduced to the minimum owing to the fact that only histograms and the results of semi-joins are redistributed across the network where their size is very small compared to the size of input relations.
The performance of this algorithm was analyzed using the BSP cost model which predicts an asymptotic optimal complexity for our algorithm even for highly skewed data.

In our future work, we will implement this algorithm and extend it to a GRID environment.

REFERENCES

Bamha, M. (2005). An optimal and skew-insensitive join and multi-join algorithm for ditributed architectures. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA'2005). 22-26 August, Copenhagen, Danemark*, volume 3588 of *Lecture Notes in Computer Science*, pages 616–625. Springer-Verlag.

Bamha, M. and Hains, G. (2000). 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*, London, United Kingdom. Springer-Verlag.

Bamha, M. and Hains, G. (2005). An efficient equi-semi-join algorithm for distributed architectures. In *Proceedings of the 5th International Conference on Com-*

putational Science (ICCS'2005). 22-25 May, Atlanta, USA, volume 3515 of *Lecture Notes in Computer Science*, pages 755–763. Springer-Verlag.

Bamha, M. and Hains, G. (September 1999). A frequency adaptive join algorithm for Shared Nothing machines. *Journal of Parallel and Distributed Computing Practices (PDCP), Volume 3, Number 3, pages 333-345.* Appears also in Progress in Computer Research, F. Columbus Ed. Vol. II, Nova Science Publishers, 2001.

Carter, J. L. and Wegman, M. N. (April 1979). Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154.

Chen, M.-S. and Yu, P. S. (1993). Combining joint and semi-join operations for distributed query processing. *IEEE Transactions on Knowledge and Data Engineering*, 5(3):534–542.

Datta, A., Moon, B., and Thomas, H. (1998). 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.

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

Gupta, A., Harinarayan, V., and Quass, D. (1995). Aggregate-query processing in data warehousing environments. In *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 358 – 369, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Li, C., Chang, K. C.-C., and Ilyas, I. F. (2005). Efficient processing of ad-hoc top-k aggregate queries in olap. Technical report, UIUCDCS-R-2005-2596, Department of Computer Science, UIUC.

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

Schneider, D. A. and DeWitt, D. J. (1989). A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data, Portland, Oregon, United States, May 1989*, pages 110–121, New York, NY, USA. ACM Press.

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

Shatdal, A. and Naughton, J. F. (1995). Adaptive parallel aggregation algorithms. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 24(2):104–114.

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

Stocker, K., Kossmann, D., Braumandl, R., and Kemper, A. (2001). Integrating semi-join-reducers into state-of-the-art query processors. In *Proceedings of the 17th International Conference on Data Engineering*, pages 575 – 584. IEEE Computer Society.

Taniar, D., Jiang, Y., Liu, K., and Leung, C. (2000). Aggregate-join query processing in parallel database systems,. In *Proceedings of The Fourth International Conference/Exhibition on High Performance Computing in Asia-Pacific Region HPC-Asia2000*, volume 2, pages 824–829. IEEE Computer Society Press.

Taniar, D. and Rahayu, J. W. (2001). Parallel processing of 'groupby-before-join' queries in cluster architecture. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid, Brisbane, Qld, Australia*, pages 178–185. IEEE Computer Society.

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

Yan, W. P. and Larson, P.-Å. (1994). Performing group-by before join. In *Proceedings of the Tenth International Conference on Data Engineering*, pages 89–100, Washington, DC, USA. IEEE Computer Society.