4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE
http://www.univ-orleans.fr/lifo

# Rapport de Recherche

## Performance Evaluation of a Parallel Algorithm for "GroupBy-Join" Queries Processing in Distributed Architectures

M. Al Hajj Hassan and M. Bamha
LIFO, Université d'Orléans

Rapport n$^o$ **RR-2007-01**

# Performance Evaluation of a Parallel Algorithm for "GroupBy-Join" Queries Processing in Distributed Architectures

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

**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 expensive 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. The performance of this algorithm is analyzed using the scalable and portable BSP (Bulk Synchronous Parallel) cost model which predicts a near optimal linear speedup even for highly skewed data. These performance predictions were also validated by the implementation results.

**Keywords:** PDBMS[1], Parallel joins, Data skew, Join product skew, GroupBy-Join queries, BSP cost model.

## 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 [10, 13, 22], because in such applications, aggregated and summarized data are more important than detailed records [8]. 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 [8]. 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 [2, 4, 15, 17].

Several parallel algorithms for evaluating "GroupBy-Join" queries were presented in the literature [18, 21], 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. In addition, the

[1] PDBMS : Parallel Database Management Systems.

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 [1, 16, 17].

The objective of this paper is to study the performance of our algorithm presented in [12] which is 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 result as in the traditional algorithms where the join operation is evaluated first and then the group-by and aggregate functions [25]. 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 [3, 4], 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 [6, 20]. The performance of this algorithm is analyzed using the BSP cost model [19] which predicts for our algorithm a near linear speedup even for highly skewed data.

The results of the practical tests assure these performance predictions and prove that it outperforms the traditional algorithms for evaluating such queries.

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. The implementation results are presented in section 5. We then conclude in section 6.

## 2 The BSP Cost Model

*Bulk-Synchronous Parallel* (BSP) cost model is a programming model introduced by L. Valiant [24] 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 [19]. 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 $h$.

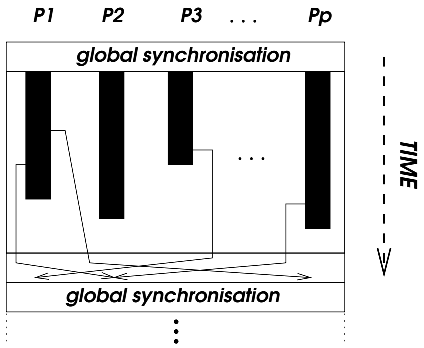


Figure 1: A BSP superstep.

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:

$$\mathrm{Time}(s) = \max_{i:processor} w_i^{(s)} + \max_{i:processor} h_i^{(s)} * g + l$$

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, $\sum_s \mathrm{Time}(s)$, 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 and Related works

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 | Query 2 |
|---|---|
| Select p.Pid, p.Pname, SUM (Quantity)<br>From PRODUCT as p, SHIPMENT as s<br>Where **p.Pid = s.Pid**<br>Group By **p.Pid, p.Pname** | Select p.Category, SUM (Quantity)<br>From PRODUCT as p, SHIPMENT as s<br>Where **p.Pid = s.Pid**<br>Group By **p.Category** |

Figure 2: Different types of "GroupBy-Join" queries.

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 [21, 22], 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. We have proposed in [11], a near linear speed-up algorithm for treating such queries.
In the contrary, group-by can not be applied before the join 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.

Taniar et al.[21] have proposed three algorithms, *Join Partition Method, Aggregate Partition Method* and *Hybrid Partition Method*, for evaluating this kind of queries. In the following three subsections we present these algorithms and we give their drawbacks.

- **Join Partition Method (JPM):**
  This algorithm is formed of four phases. The first phase is the *data partitioning phase* where the two relations to be joined are partitioned into $N$ buckets, by applying a partitioning method such as range partitioning on the join attribute ($N$ is the number of processors). After that the buckets will be redistributed on the different processors.
  In the second phase, *join and local aggregate*, the join operation is performed in parallel on each processor using any sequential join algorithm, and after that the aggregate operation is applied locally on each processor.
  The third phase is a *re-distribution phase* where a partitioning methods is applied on the group-by attribute in order to send tuples having the same values of this attribute to the same processor.
  The final phase is the *global aggregation phase* where each processor applies the aggregate and group-by operations on the received tuples in order to find the final result.

  The main drawback of this algorithm lies in applying the partition or hash functions on the join attribute of the base relations because this may cause a skew in the load of different processors if we have a skew in the distribution of the values of the join attribute [1, 16, 17]. We may also have a skew in the result of the join (JPS) which also affects the performance of this algorithm because the join result is the input of the local aggregation step.

- **Aggregate Partition Method (APM):**
  This is a two phase algorithm where in the first phase, *data partitioning and broadcasting*, the table that contains the group-by attribute will be partitioned into $N$ buckets by applying partitioning method on this attribute. After that these buckets are distributed on the processors and the second table is broadcasted to all processors.
  The second phase is the *join and aggregate phase* where each processor performs the join of the received fragments of the first relations with the entire second table. After that, the aggregate and group-by operations are computed on each processor.

  In this algorithm, we do not need to redistribute the join result in order to apply the group-by and aggregate function because all tuples having the same value of the group-by attribute will be found on the same processor after the redistribution of the first phase. But it introduces other expensive costs, especially communication costs because the relation that does not contain the group-by attribute must be fully replicated on all the processors which makes the algorithm inextensible. In addition it may suffer from high input/output costs due to hash table overflow in the second phase.

  The performance tests presented in [23] show that the performance of the *aggregate partition method* is better than that of the *join partition method* when the number of produced joins is big, but not when it is relatively small. The speed-up tests also show that both algorithms have a non-linear acceleration and that skew has a great impact on their performance.

- **Hybrid Partition Method (HPM):**
  In this method the $N$ processors are divided into $m$ logical clusters, each containing $N/m$ processors. In the first step, the table that contains the group-by attribute is partitioned into the different clusters using the *APM* method. In the second step, within each cluster, the received fragment of the first table and the broadcasted table are partitioned by the join attributes using the *JPM* method.

  In this algorithm, the table that contains the group-by attribute is purely partitioned on all the processors. This is not the case for the second table which is to some degree replicated [21] hence the communication costs stays high. In addition, it does not solve the problem of data skew because the partitioning method is applied, as in *JPM*, on the tuples of the base relations which may be highly skewed.

The algorithm that we propose in this paper overcomes the drawbacks of the three algorithms presented in this section. In this algorithm, we succeeded to partially evaluate the aggregate functions before redistributing the tuples using histograms, thus reducing the communication cost as much as possible. In addition, we do not fully materialize the join result before applying the aggregate and join operations hence reducing the input/output cost and it is insensitive to skewness.

# 4 GroupBy-Join Queries: "OGJA" 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[2] 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 $T_i$,
- $Hist^w_i(T)$ 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 $T_i$,
- $AGGR^w_{f,u}(T)$ [3] 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 $w$. $AGGR^w_{f,u}(T)$ 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),
- $AGGR^w_{f,u}(T_i)$ denotes the result of applying the aggregate function on the attribute $u$ of relation $T_i$,

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

- $AGGR^w_{f,u,i}(T)$ is processor $i$'s fragment of the result of applying the aggregate function on $T$,
- $AGGR^w_{f,u}(T)(v)$ 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 $T$,
- $AGGR^w_{f,u}(T_i)(v)$ 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 [4]. We will give an upper bound of the execution time of each superstep using BSP cost model. 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.

**Phase 1: Creating local histograms**

In this phase, the local histograms $Hist^{x,y}(R_i)(i = 1,...,p)$ of blocks $R_i$ are created in parallel by a scan of the fragment $R_i$ on processor $i$ in time $c_{i/o} * \max_{i=1,...,p} |R_i|$ where $c_{i/o}$ is the cost of writing/reading a page of data from disk.
In addition, the local fragments $AGGR^{x,z}_{f,u}(S_i)(i = 1,...,p)$ 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 $c_{i/o} * \max_{i=1,...,p} |S_i|$. (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 $AGGR^{x,z}_{f,u}(S_i)$ 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 $AGGR^{x,z}_{f,u}(S_i)$. So while creating $Hist^{x,y}(R_i)$ (resp. $AGGR^{x,z}_{f,u}(S_i)$), 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. $AGGR^{x,z}_{f,u}(S_i)$) for $i = 1,...,p$.
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 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 *Algorithm 1* to create $Hist^{x,y}(R_i)$ and $Hist'^{x}(R_i)$. A similar algorithm is used to create $AGGR^{x,z}_{f,u}(S_i)$ and $Hist'^{x}(S_i)$ but instead of computing the frequency of $(x, y)$ to create $Hist^{x,y}(R_i)$, we apply here the aggregate function on $(x, z)$ to create $AGGR^{x,z}_{f,u}(S_i)$.

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

**Phase 2: Local semi-joins computation**

In order to minimize the communication cost, only tuples of $Hist^{x,y}(R)$ and $AGGR^{x,z}_{f,u}(S)$ that will be present in the join result are redistributed. These tuples are effectively the result of the following local semi-joins: $\overline{Hist}^{x,y}(R_i) = Hist^{x,y}(R_i) \ltimes AGGR^{x,z}_{f,u}(S)$ and $\overline{AGGR}^{x,z}_{f,u}(S_i) = AGGR^{x,z}_{f,u}(S_i) \ltimes Hist^{x,y}(R)$, but it is expensive to duplicate $AGGR^{x,z}_{f,u}(S)$ and $Hist^{x,y}(R)$ on all the processors. Hence to override this problem, we use proposition 2 presented in [3] to compute these semi-joins, 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 $Hist'^{x}(S_i)$ on each processor $i$.
The cost of this phase is [3]:

[4] Review Appendix A for a detailed example of using this algorithm

**Algorithm 1: Computing $Hist^{x,y}(R_i)$ and $Hist'^{x}(R_i)$**

```
Par (on each node in parallel) i = 1, ..., p
  Hist^{x,y}(R_i) = NULL   (This instruction creates a B-tree to store histogram's entries.)
  Hist'^x(R_i) = NULL
  For every tuple t of R_i do
    If Hist^{x,y}(R_i)(t.x, t.y) = 0 Then
      Insert a new tuple (t.x, t.y, 1) into Hist^{x,y}(R_i)
      freq_1 = Hist'^x(R_i)(t.x)
      If freq_1 ≠ 0 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
    Else
      Increment the frequency of (t.x, t.y) in Hist^{x,y}(R_i)
    EndIf
  EndFor
EndPar
```

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

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

We recall (cf. to proposition 1 in [3]) 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 necessary time to compute the global histograms $Hist^x_{i=1,...,p}(R)$ and $Hist^x_{i=1,...,p}(S)$, respectively starting from the local histograms $Hist^x(R_i)$ and $Hist^x(S_i)$ where $i = 1, ..., p$, but 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 above cost still holds for computing $Hist'^x_{i=1,...,p}(R)$ and $Hist'^x_{i=1,...p}(S)$.
During semi-join computation, we store on each processor $i$, an extra information called $index(d) \in \{1, 2, 3\}$ for each value $d \in Hist'^x_i(R) \cap Hist'^x_i(S)$. 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 [2, 4], i.e. $f_0 = p*log(p)$. On each processor $i$, for a given value $d \in Hist'^x_i(R) \cap Hist'^x_i(S)$ [5],

- the value $index(d) = 3$, means that the frequency of tuples of relations $Hist^{x,y}_i(R)$ and $AGGR^{x,z}_{f,u,i}(S)$

[5]The intersection of $Hist'^x_i(R)$ and $Hist'^x_i(S)$ is found while computing the semi-joins (c.f proposition 2 presented in [3])

associated to value $d$ are less than the threshold frequency. (i.e. $Hist_i^{'x}(R)(d) < f_0$ and $Hist_i^{'x}(S)(d) < f_0$),

- the value $index(d) = 2$, means that $Hist_i^{'x}(S)(d) \geq f_0$ and $Hist_i^{'x}(S)(d) > Hist_i^{'x}(R)(d)$,
- the value $index(d) = 1$, means that $Hist_i^{'x}(R)(d) \geq f_0$ and $Hist_i^{'x}(R)(d) \geq Hist_i^{'x}(S)(d)$.

Note that unlike the algorithms presented in [18, 21] where both relations $R$ and $S$ are redistributed, we will only redistribute $Hist^{x,y}(R_i) \ltimes AGGR_{f,u}^{x,z}(S)$ and $AGGR_{f,u}^{x,z}(S_i) \ltimes Hist^{x,y}(R)$ 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 $\overline{AGGR}_{f,u}^{x,z}(S_i)$ 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

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

where:

- All the tuples of $\overline{Hist}^{(1)x,y}(R_i)$ (resp. $\overline{AGGR}_{f,u}^{(1)x,z}(S_i)$) 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. $\overline{AGGR}_{f,u}^{(2)x,z}(S_i)$) 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 $\overline{AGGR}_{f,u}^{(3)x,z}(S_i)$ 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 $\overline{Hist}^{(1)x,y}(R_i)$ and $\overline{AGGR}_{f,u}^{(1)x,z}(S_i)$ 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 $\overline{AGGR}_{f,u}^{(3)x,z}(S_i)$ (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**

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 [4, 2, 1].

**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. $index(d) = 3$),

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

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 $Hist_i^{(1,2)'x}(R \bowtie S)$. 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 $Hist^{'x}(S)$.

---

**if** $\big(Hist^{'x}(R)(d) mod(p) = 0\big)$ **then**

`each processor` $j$ `will hold a block of size` $block_j(d) = \dfrac{Hist^{'x}(R)(d)}{p}$ `of tuples of value` $d$.

**else**

**begin**

`Pick a random value` $j_0$ `between` $0$ `and` $(p-1)$

**if** $\Big($`processor's index` $j$ `is between` $j_0$ `and` $j_0 + \big(Hist^{'x}(R)(d)\, mod\, p\big)\Big)$ **then**

`the processor of index` $j$ `will hold a block of size:` $block_j(d) = \lfloor \dfrac{Hist^{'x}(R)(d)}{p} \rfloor + 1$

**else**

`the processor of index` $j$ `will hold a block of size:` $block_j(d) = \lfloor \dfrac{Hist^{'x}(R)(d)}{p} \rfloor$

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

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

Each 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 $\alpha$ (resp. $\beta$) 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:

$i := 1$;   $j := \alpha + 1$;
**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**.

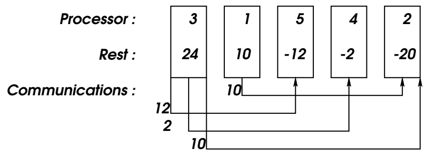


Figure 2: Sending orders as a function of *Rest* values.

Figure 2 gives an example of the value *Rest* associated 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 $\overline{AGGR}_{f,u}^{(3)x,z}(S_i)$ (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 $\overline{AGGR}_{f,u}^{x,z}(S_i)$ will be redistributed. Redistribution is then performed in time: $Time_{4.b} = O\Big(g * \big(|\overline{Hist}^{x,y}(R_i)| + |\overline{AGGR}_{f,u}^{x,z}(S_i)|\big) + l\Big)$.
Thus the total cost of the redistribution phase is the sum of the costs of the above two steps:

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

We mention that we only redistribute the tuples of the semi-joins $\overline{Hist}^{x,y}(R_i)$ and $\overline{AGGR}_{f,u}^{x,z}(S_i)$ where $|\overline{Hist}^{x,y}(R_i)|$ and $|\overline{AGGR}_{f,u}^{x,z}(S_i)|$ 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 $|AGGR_{f,u}^{x,z}(S)|$. 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 [1], 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 *Algorithm 2*, 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 $AGGR^{y,z}_{f,u}((R \bowtie S)_i)$ that holds the local results of applying the aggregate function on every group of tuples having the value of the couple of attributes $(y, z)$. $AGGR^{y,z}_{f,u}((R \bowtie S)_i)$ has the form $(y, z, v)$ where $y$ and $z$ are the group-by attributes and $v$ is the result of the aggregate function.

The cost of applying this algorithm is:

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

So the total cost of this phase is:

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

**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, $AGGR^{y,z}_{f,u}((R \bowtie S)_i)$, 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 [3]) in redistributing $AGGR^{y,z}_{f,u}((R \bowtie S)_i)$.

**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 [5],
- *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 [9]. 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:

**Algorithm 2:** Join operation and local computation of aggregate function.

**Par** (on each node in parallel) $i = 1, ..., p$
  $AGGR^{y,z}_{f,u}((R \bowtie S)_i) = \text{NULL};$ [7]
  **For every tuple** $t$ **of relation** $\overline{Hist}^{(1)x,y}(R_i)$ **do**
    **For every entry** $v_1 = \overline{AGGR}^{(2)x,z}_{f,u}(S)(t.x, z)$ **do**
      $v_2 = AGGR^{y,z}_{f,u}((R \bowtie S)_i)(t.y, z);$
      **If** $v_2 \neq 0$ **Then**
        Update $AGGR^{y,z}_{f,u}((R \bowtie S)_i)(t.y, z) = F(n, v_1, v_2)$ where $n$ is the frequency of $t.x$ in $\overline{Hist}^{(1)x,y}(R_i)$ and $F()$ is the aggregate function;
      **Else**
        Insert a new tuple $(t.y, z, f(n, v_1))$ into the histogram $AGGR^{y,z}_{f,u}((R \bowtie S)_i);$
      **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** $v_1 = \overline{Hist}^{(2)x,y}(R)$ **do**
      $v_2 = AGGR^{y,z}_{f,u}((R \bowtie S)_i)(t.y, z);$
      **If** $v_2 \neq 0$ **Then**
        Update $AGGR^{y,z}_{f,u}((R \bowtie S)_i)(t.y, z) = F(t.f_v, v_1, v_2)$
      **Else**
        Insert a new tuple $(t.y, z, f(t.f_v, v_1))$ into the histogram $AGGR^{y,z}_{f,u}((R \bowtie S)_i);$
      **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**
      $v_2 = AGGR^{y,z}_{f,u}((R \bowtie S)_i)(t.y, z);$
      **If** $v_2 \neq 0$ **Then**
        Update $AGGR^{y,z}_{f,u}((R \bowtie S)_i)(t.y, z) = F(n, v_1, v_2)$ where $n$ is the frequency of $t.x$ in $\overline{Hist}^{(3)x,y}(R_i)$
      **Else**
        Insert a new tuple $(t.y, z, f(n, v_1))$ into the histogram $AGGR^{y,z}_{f,u}((R \bowtie S)_i)$
      **EndIf**
    **EndFor**
  **EndFor**
**EndPar**

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

**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:

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

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 Performance Evaluation

In order to validate our theoretical predictions, we implemented our algorithm on a Shared Nothing machine formed of 8 processors using MySQL-5.0.26-log database server and MPI-2 as a Message Passing Interface for communications. In the sensitivity analysis presented in this section, the performance of our algorithm is compared with that of *Join Partition method (JPM)* presented in [21] where we performed speed-up tests and we studied the effect of the data skew on the performance of both algorithms.

The theoretical cost model presented in the fourth section of this paper and the experimental analysis assure that the gain of using our algorithm increases when used on a *SN* machine formed of a number of processors higher than that we have used in our tests.

The skewness in the frequencies of the *Join Attribute Values (AVS)* is an important factor that may affect the performance of the parallel database algorithms. Hence, to study the effect of the AVS on the performance of both algorithms, we used the *Zipf* [26, 7, 14] model to determine the frequencies of the join attribute values. Using this model, in a relation $R$ with a domain $\{1, 2, \dots, D\}$ of distinct values for its join attribute, the $i^{th}$ distinct value has a number of tuples given by the following expression :

$$\|v_i\| = \frac{\|R\|}{i^z * \sum_{j=1}^{D} \frac{1}{j^z}}$$

where $z$ is the skew factor. When $z = 0$ the distribution is uniform and when $z \geq 1$ it becomes highly skewed.

In the performance tests presented in this paper, we used two relations $R$ and $S$ formed of $100K$ and 1 million tuples respectively with a domain formed of $100K$ distinct values for the join attribute in both relations. The aggregate attribute is an attribute of $S$.

We have measured the JPS as the percentage of the maximum deviation of the local join result size with respect to the average over all processors.

## 5.1 Speed-up test

To perform the speed-up test, we varied the number of processors from 1 to 8 and the *Zipf* skew factor was fixed to 0.5 which represents a weak degree of skewness.
The total join size reached 5 million tuples while the final result was formed of approximately 10244 groups. Graph ($a$) of Fig.3 shows that our algorithm has a near optimal linear speed-up and it outperforms the *Join Partition Method* (JPM) even for this low value of skewness.
The difference in the performance of the two algorithms is due to the following reasons:

- In JPM algorithm all the tuples of the base relations are redistributed even if they do not participate in the join operation. In the contrary, in our algorithm, we partially apply the aggregate functions on the base relations in the first phase hence decreasing their sizes, after that only the tuples of the aggregated relations that effectively participate in the join result will be redistributed. So, the communication cost is reduced to minimum.

- In *JPM*, the result of the join operation is materialized before applying the aggregate and the group-by operations which results in high input/output costs, this is not the case in our algorithm where, in phase 5, we locally compute the aggregate function on each processor without fully materializing the intermediate results of the join operation. In addition, we can see in Fig.3(b) that the *JPS* value in the *Join Partition Method* reached 23.94% even for this low factor of *AVS* while its maximum value in our algorithm was 7.42% and as we mentioned above there is a great difference between the load of the processors after the join phase in the two algorithms hence the JPS percentage in our algorithm is negligible.

- Due to this difference in the join result size, the cost of computing the global result of the aggregate function in the *JPM* method is very much higher than in our algorithm.

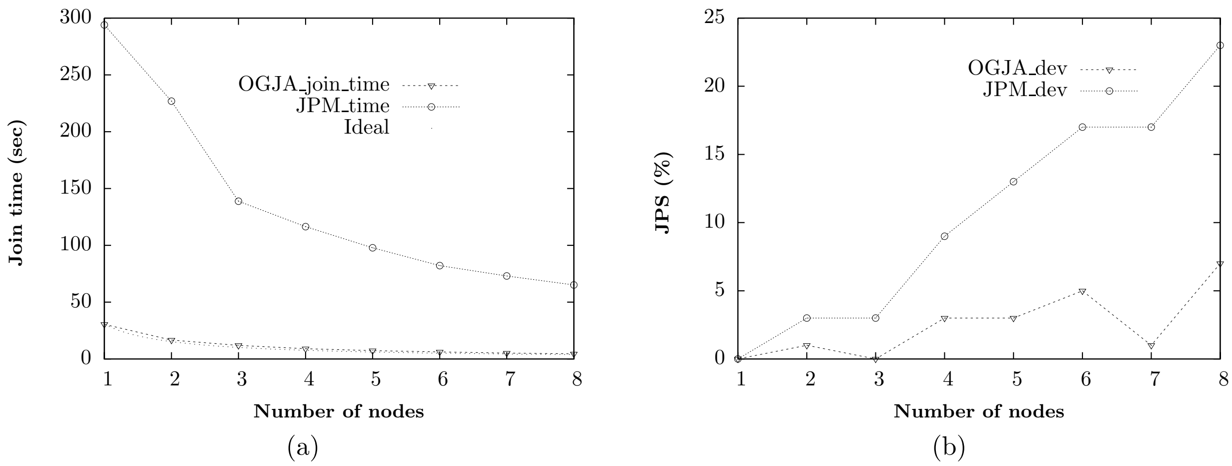


Figure 3: Speedup performance for small attribute skew.

### 5.2 Attribute value skew test

As we have said earlier, the AVS is an important factor in database query evaluation in parallel system, so in order to study its effect on both algorithms we fixed the number of processors to 8 and the AVS of relation $S$ was varied from 0 to 1.8.

Graph (a) in Fig. 4 shows that our algorithm outperforms the JPM algorithm and that its processing time is balanced while varying the skew value. We can also deduce from graph (b) that the JPS is approximately negligible in our algorithm while it is important in JPM algorithm.

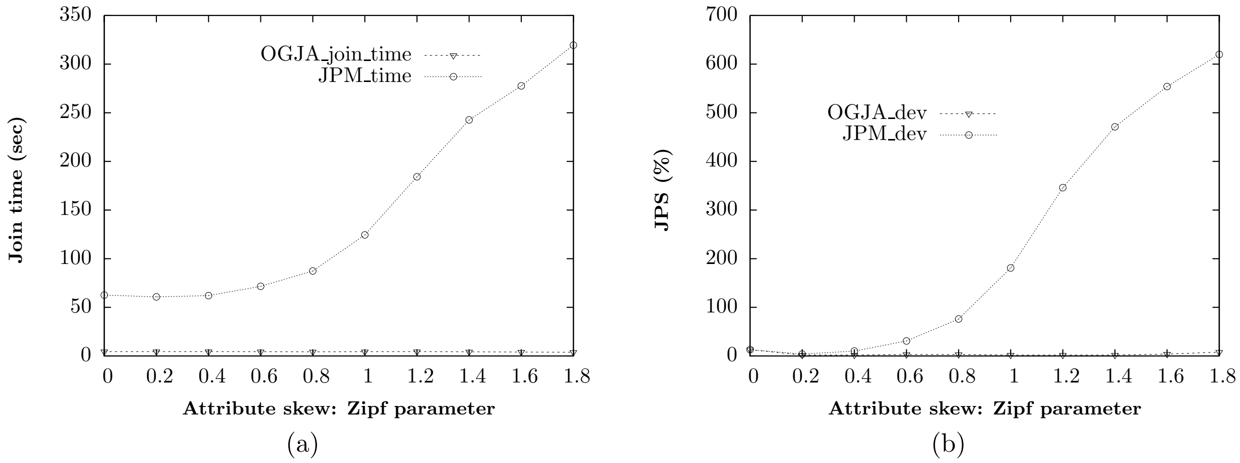


Figure 4: Effect of the attribute value skew.

### 5.3 Summary of the performance tests

The presented performance tests and all the other tests that we have performed prove that our algorithm has a near linear speed-up, and it is unaffected by the AVS and has a near perfect balancing properties for the intermediate join phase.

## 6 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 BSP cost analysis and the implementation results assure that our algorithm has a near optimal linear complexity even for highly skewed data.

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

# Appendix A

In this appendix, we present a detailed example of evaluating "GroupBy-Join" queries using *"OGJA"* algorithm.
The query can be expressed in $SQL$ as:

SELECT $R.y, S.z, SUM(S.u)$
FROM $R, S$
WHERE $R.x = S.x$
GROUP BY $R.y, S.z$

where $x$ is the join attribute of relations $R$ and $S$, $y$ and $z$ are the group-by attributes of $R$ and $S$ respectively and the aggregate function $SUM$ is applied on the attribute $u$ of relation $S$.

The algorithm proceeds in the following manner, where we assume that relations $R$ and $S$ are partitioned on 3 processors:

**Phase 1: Creating local histograms**
In the first step, we create $Hist^{x,y}(R_i)$, in parallel on each processor $i$, as shown in figure 5. We also create, on the fly, the local histograms $Hist'^{x}(R_i)$ (fig. 6) in parallel on each processor $i$.
The aggregate function $SUM$ is also applied on the attribute $u$ of every group of tuples of $S_i$ having the

**$Hist^{x,y}(R_1)$**

| X | Y | Freq |
|---|---|---|
| 1 | 1 | 100 |
| 1 | 3 | 300 |
| 1 | 7 | 200 |
| 1 | 10 | 50 |
| 1 | 15 | 40 |
| 3 | 3 | 40 |
| 3 | 4 | 100 |
| 3 | 7 | 100 |
| 4 | 1 | 150 |
| 4 | 3 | 100 |
| 4 | 4 | 200 |
| 4 | 10 | 50 |
| 6 | 2 | 100 |
| 6 | 3 | 50 |
| 6 | 12 | 50 |

**$Hist^{x,y}(R_2)$**

| X | Y | Freq |
|---|---|---|
| 1 | 3 | 200 |
| 1 | 10 | 100 |
| 2 | 3 | 200 |
| 2 | 4 | 50 |
| 2 | 5 | 50 |
| 2 | 8 | 100 |
| 2 | 10 | 100 |
| 2 | 13 | 20 |
| 2 | 15 | 30 |
| 2 | 20 | 50 |
| 4 | 10 | 100 |
| 4 | 15 | 200 |
| 7 | 2 | 100 |
| 7 | 3 | 50 |
| 7 | 10 | 13 |
| 7 | 23 | 100 |

**$Hist^{x,y}(R_3)$**

| X | Y | Freq |
|---|---|---|
| 1 | 3 | 100 |
| 1 | 10 | 50 |
| 1 | 13 | 20 |
| 1 | 16 | 30 |
| 1 | 18 | 100 |
| 1 | 20 | 50 |
| 1 | 25 | 50 |
| 4 | 5 | 80 |
| 4 | 10 | 80 |
| 4 | 12 | 100 |
| 4 | 14 | 100 |
| 4 | 20 | 50 |
| 4 | 25 | 50 |
| 7 | 2 | 150 |
| 7 | 3 | 140 |

Figure 5: $Hist^{x,y}(R_i)$ of local blocks $R_i$.

**$Hist'^{x}(R_1)$**

| X | Freq |
|---|---|
| 1 | 5 |
| 3 | 3 |
| 4 | 4 |
| 6 | 3 |

**$Hist'^{x}(R_2)$**

| X | Freq |
|---|---|
| 1 | 2 |
| 2 | 8 |
| 4 | 2 |
| 7 | 4 |

**$Hist'^{x}(R_3)$**

| X | Freq |
|---|---|
| 1 | 7 |
| 4 | 6 |
| 7 | 2 |

Figure 6: local histograms $Hist'^{x}(R_i)$.

same value of $(x, z)$ in order to create in parallel $AGGR^{x,z}_{f,u}(S_i)$ (fig. 7) on each processor $i$. At the same time, we create on the fly $Hist'^{x}(S_i)$ (fig. 8) locally on each processor $i$.

$\mathbf{AGGR^{x,z}_{f,u}(S_1)}$

| X | Z | SUM |
|---|---|---|
| 1 | 2 | 200 |
| 1 | 10 | 150 |
| 1 | 15 | 300 |
| 3 | 10 | 50 |
| 3 | 13 | 600 |
| 3 | 15 | 200 |
| 4 | 1 | 200 |
| 4 | 4 | 140 |
| 4 | 12 | 250 |
| 4 | 13 | 600 |
| 4 | 14 | 100 |
| 4 | 18 | 20 |
| 4 | 20 | 100 |
| 4 | 21 | 200 |
| 11 | 4 | 150 |
| 11 | 10 | 20 |
| 11 | 18 | 200 |
| 11 | 20 | 100 |

$\mathbf{AGGR^{x,z}_{f,u}(S_2)}$

| X | Z | SUM |
|---|---|---|
| 1 | 10 | 200 |
| 1 | 12 | 130 |
| 3 | 12 | 100 |
| 3 | 15 | 200 |
| 4 | 12 | 50 |
| 4 | 13 | 100 |
| 4 | 14 | 50 |
| 4 | 20 | 210 |
| 4 | 25 | 300 |
| 4 | 26 | 100 |
| 4 | 27 | 140 |
| 4 | 30 | 50 |
| 4 | 31 | 20 |
| 4 | 34 | 30 |
| 4 | 35 | 200 |
| 4 | 39 | 10 |
| 4 | 40 | 12 |
| 4 | 48 | 10 |
| 12 | 20 | 60 |
| 12 | 21 | 60 |

$\mathbf{AGGR^{x,z}_{f,u}(S_3)}$

| X | Z | SUM |
|---|---|---|
| 1 | 12 | 200 |
| 1 | 18 | 150 |
| 4 | 12 | 100 |
| 4 | 13 | 50 |
| 4 | 20 | 100 |
| 4 | 25 | 150 |
| 4 | 30 | 125 |
| 4 | 33 | 50 |
| 2 | 10 | 30 |
| 2 | 15 | 40 |
| 2 | 21 | 100 |
| 2 | 25 | 150 |
| 2 | 30 | 200 |
| 13 | 1 | 220 |
| 13 | 10 | 140 |
| 13 | 15 | 200 |
| 13 | 20 | 250 |
| 13 | 25 | 300 |
| 13 | 29 | 350 |

Figure 7: $AGGR^{x,z}_{f,u}(S_i)$ related to $S_i$.

$\mathbf{Hist'^{x}(S_1)}$

| X | Freq |
|---|---|
| 1 | 3 |
| 3 | 3 |
| 4 | 8 |
| 11 | 4 |

$\mathbf{Hist'^{x}(S_2)}$

| X | Freq |
|---|---|
| 1 | 2 |
| 3 | 2 |
| 4 | 14 |
| 12 | 2 |

$\mathbf{Hist'^{x}(S_3)}$

| X | Freq |
|---|---|
| 1 | 2 |
| 4 | 6 |
| 2 | 5 |
| 13 | 6 |

Figure 8: local histograms $Hist'^{x}(S_i)$.

**Phase 2: local semi-join computation**

In this phase we compute $\overline{Hist}^{x,y}(R_i) = Hist^{x,y}(R_i) \ltimes AGGR^{x,z}_{f,u}(S)$ and $\overline{AGGR}^{x,z}_{f,u}(S_i) = AGGR^{x,z}_{f,u}(S_i) \ltimes Hist^{x,y}(R)$ using proposition 2 presented in [3].
First of all we need to evaluate the global histograms $Hist'^{x}_i(R)$ and $Hist'^{x}_i(S)$ (fig. 8) for $i \in \{1,2,3\}$. Hence, the tuples of $Hist'^{x}(R_i)$ and $Hist'^{x}(S_i)$ are redistributed by applying a hash function on the values of the attribute $x$ in order to send tuples having the same value of $x$ to the same processor. Here we use the hash function $(x \ mod \ 3)+1$.

After that we create, in parallel on each processor $i$, $Hist'^{x}_i(R \bowtie S)$ (fig. 9) which is simply the intersection of $Hist'^{x}_i(R)$ and $Hist'^{x}_i(S)$. At the same time we store the value of $index(d)$ for each value $d$ of $x$ [8].

Now we know the values of $x$ that will be present in the result of the join operation, so to reduce the communication costs only tuples of $Hist^{x,y}(R_i)_{i\in\{1,2,3\}}$ and $AGGR^{x,z}_{f,u}(S_i)_{i\in\{1,2,3\}}$ having these values of $x$ must be redistributed for farther treatment.
To this end, we compute on each processor $j$, $\overline{Hist}^{j'x}(R_i)$ and $\overline{Hist}^{j'x}(R_i)$ for $i \in \{1,2,3\}$, where $\overline{Hist}^{j'x}(R_i) = Hist^{j'x}(R_i) \cap Hist'^{x}_j(R \bowtie S)$ and $\overline{Hist}^{j'x}(S_i) = Hist^{j'x}(S_i) \cap Hist'^{x}_j(R \bowtie S)$. As shown in fig. 11, $Hist^{j'x}(R_i)$

[8] In this example, we consider that the value of the threshold frequency $f_0$ is 6.

$Hist_1'^x(R)$

| X | Freq |
|---|---|
| 3 | 3 |
| 6 | 3 |

$Hist_2'^x(R)$

| X | Freq |
|---|---|
| 1 | 14 |
| 4 | 12 |
| 7 | 6 |

$Hist_3'^x(R)$

| X | Freq |
|---|---|
| 2 | 8 |

$Hist_1'^x(S)$

| X | Freq |
|---|---|
| 3 | 5 |
| 12 | 2 |

$Hist_2'^x(S)$

| X | Freq |
|---|---|
| 1 | 7 |
| 4 | 28 |
| 13 | 6 |

$Hist_3'^x(S)$

| X | Freq |
|---|---|
| 2 | 5 |
| 11 | 4 |

Figure 9: global histograms $Hist_i'^x(R)$ and $Hist_i'^x(S)$

$\mathbf{Hist_1'^x(R \bowtie S)}$

| $X$ | $Freq$ | $index(d)$ |
|---|---|---|
| 3 | 15 | 3 |

$\mathbf{Hist_2'^x(R \bowtie S)}$

| $X$ | $Freq$ | $index(d)$ |
|---|---|---|
| 1 | 98 | 1 |
| 4 | 336 | 2 |

$\mathbf{Hist_3'^x(R \bowtie S)}$

| $X$ | $Freq$ | $index(d)$ |
|---|---|---|
| 2 | 40 | 1 |

Figure 10: partitions of the histogram of $R \bowtie S$.

is the partition of $Hist'^x(R_i)$ which was sent to processor $j$ (this also applies on $Hist'^x(S_i)$).
After computing $\overline{Hist}^{j'x}(R_i)$ and $\overline{Hist}^{j'x}(S_i)$, we perform a communication step where each processor $j$ sends each fragment $\overline{Hist}^{j'x}(R_i)$ and $\overline{Hist}^{j'x}(S_i)$ to processor $i$. After redistribution each processor $i$ merges the received tuples in order to create $\overline{Hist}'^x(R_i)$ which represents the values of $x$ that participate in the join operation. Now we can compute on each processor $i$ in parallel $\overline{Hist}^{x,y}(R_i)$ (fig. 12) and $\overline{AGGR}_{f,u}^{x,z}(S_i)$ (fig. 13) where we do not need to create $AGGR_{f,u}^{x,z}(S)$ and $Hist^{x,y}(R)$ because $\overline{Hist}^{x,y}(R_i)$ is simply $Hist^{x,y}(R_i) \cap \big(\bigcup_{j=1}^{3} \overline{Hist}^{j'x}(R_i)\big) = Hist^{x,y}(R_i) \cap \overline{Hist}'^x(R_i)$ and $\overline{AGGR}_{f,u}^{x,z}(S_i) = AGGR_{f,u}^{x,z}(S_i) \cap \overline{Hist}'^x(S_i)$.

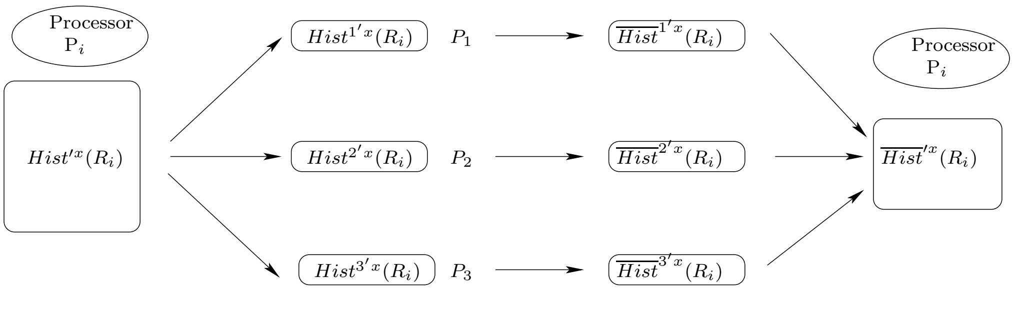


Figure 11: computation of $\overline{Hist}'^x(R_i)$.

**Phase 3: Creating the communication templates**

In this phase we need to create the communication templates in order to redistribute the tuples of $\overline{Hist}^{x,y}(R_i)_{i\in\{1,2,3\}}$ and $\overline{AGGR}_{f,u}^{x,z}(S_i)_{i\in\{1,2,3\}}$ in a manner that helps us to avoid the problems caused by AVS and JPS.
So on each processor $i$, we partition $Hist_i'^x(R \bowtie S)$ into $Hist_i^{(1,2)'x}(R \bowtie S)$ associated to high frequencies

$\overline{\textbf{Hist}}^{\textbf{x,y}}(\mathbf{R_1})$

| X | Y | Freq |
|---|---|---|
| 1 | 1 | 100 |
| 1 | 3 | 300 |
| 1 | 7 | 200 |
| 1 | 10 | 50 |
| 1 | 15 | 40 |
| 3 | 3 | 40 |
| 3 | 4 | 100 |
| 3 | 7 | 100 |
| 4 | 1 | 150 |
| 4 | 3 | 100 |
| 4 | 4 | 200 |
| 4 | 10 | 50 |

$\overline{\textbf{Hist}}^{\textbf{x,y}}(\mathbf{R_2})$

| X | Y | Freq |
|---|---|---|
| 1 | 3 | 200 |
| 1 | 10 | 100 |
| 2 | 3 | 200 |
| 2 | 4 | 50 |
| 2 | 5 | 50 |
| 2 | 8 | 100 |
| 2 | 10 | 100 |
| 2 | 13 | 20 |
| 2 | 15 | 30 |
| 2 | 20 | 50 |
| 4 | 10 | 100 |
| 4 | 15 | 200 |

$\overline{\textbf{Hist}}^{\textbf{x,y}}(\mathbf{R_3})$

| X | Y | Freq |
|---|---|---|
| 1 | 3 | 100 |
| 1 | 10 | 50 |
| 1 | 13 | 20 |
| 1 | 16 | 30 |
| 1 | 18 | 100 |
| 1 | 20 | 50 |
| 1 | 25 | 50 |
| 4 | 5 | 80 |
| 4 | 10 | 80 |
| 4 | 12 | 100 |
| 4 | 14 | 100 |
| 4 | 20 | 50 |
| 4 | 25 | 50 |

Figure 12: $\overline{Hist}^{x,y}(R_i)$, $i \in \{1, 2, 3\}$.

(i.e. $index(d) = 1$ or 2) and $Hist_i^{(3)'x}(R \bowtie S)$ associated to low frequencies (i.e. $index(d) = 3$).
Tuples of $\overline{Hist}^{x,y}(R_i)_{i\in\{1,2,3\}}$ and $\overline{AGGR}^{x,z}_{f,u}(S_i)_{i\in\{1,2,3\}}$ associated to low frequencies are redistributed by applying a hash function because they do not have any effect on AVS and JPS.
However, we need a special treatment for tuples associated to high frequencies, where we must evenly redistribute these tuples on the different processors.
In our example, in processor 2 the value of $index(d)$ is 1 for x=1, hence the frequency of attribute $x$ having value 1 in $\bigcup_{i=1}^{3} \overline{Hist}^{x,y}(R_i)$ is higher than that in $\bigcup_{i=1}^{3} \overline{AGGR}^{x,z}_{f,u}(S_i)$. So each processor must have after redistribution approximately $\frac{14}{3}$ tuples of $\bigcup_{i=1}^{3} \overline{Hist}^{x,y}(R_i)$ having value 1 of x where 14 is the frequency of 1 and 3 is the number of processors. However, Tuples of $\bigcup_{i=1}^{3} \overline{AGGR}^{x,z}_{f,u}(S_i)$ having value 1 of $x$ must be duplicated on all the processors in order to apply the join operation.

**Phase 4: Data redistribution**
At the end of phase 3, each processor $i$ knows the number of tuples for every one of its local values of $x$ in $Hist_i^{(1,2)'x}(R \bowtie S)$ that each processor $j$ actually has (i.e. it knows the frequency of each value of $x$ in $Hist^{x,y}(R_j)$ and $AGGR^{x,z}_{f,u}(S_j)$ where $j \in \{1, 2, 3\}$). It also knows the number of tuples that each processor $j$ must have after redistribution in order to balance the load of all processors. So it sends *sending orders* to processors that have more tuples than needed specifying the number of tuples that they must send to processors that have lack in the number of tuples.

For example, value 2 of x belongs to $Hist_3^{(1,2)'x}(R \bowtie S)$ and its value of $index(d)$ is 1. Processor 3 knows that processors 1 and 3 do not have any tuple having value 2 of x in their local partitions of $\overline{Hist}^{x,y}(R)$, and that processor 2 has 8 tuples associated to this value. Hence each processor must have approximately $\frac{8}{3}$ tuples after redistribution. So it will ask processor 2 to send 3 tuples to processor 1 and 2 tuples to processor 3 having value 2 of $x$.

Now each processor $i$ redistributes $\overline{Hist}^{x,y}(R_i)$ and $\overline{AGGR}^{x,z}_{f,u}(S_i)$ using the *sending orders* that it has received. As we mentioned in phase 3, tuples associated to low frequency of $x$ are redistributed by hashing.

$\overline{\mathbf{AGGR}}^{\mathbf{x,z}}_{\mathbf{f,u}}(\mathbf{S_1})$

| X | Z | SUM |
|---|---|---|
| 1 | 2 | 200 |
| 1 | 10 | 150 |
| 1 | 15 | 300 |
| 3 | 10 | 50 |
| 3 | 13 | 600 |
| 3 | 15 | 200 |
| 4 | 1 | 200 |
| 4 | 4 | 140 |
| 4 | 12 | 250 |
| 4 | 13 | 600 |
| 4 | 14 | 100 |
| 4 | 18 | 20 |
| 4 | 20 | 100 |
| 4 | 21 | 200 |

$\overline{\mathbf{AGGR}}^{\mathbf{x,z}}_{\mathbf{f,u}}(\mathbf{S_2})$

| X | Z | SUM |
|---|---|---|
| 1 | 10 | 200 |
| 1 | 12 | 130 |
| 3 | 12 | 100 |
| 3 | 15 | 200 |
| 4 | 12 | 50 |
| 4 | 13 | 100 |
| 4 | 14 | 50 |
| 4 | 20 | 210 |
| 4 | 25 | 300 |
| 4 | 26 | 100 |
| 4 | 27 | 140 |
| 4 | 30 | 50 |
| 4 | 31 | 20 |
| 4 | 34 | 30 |
| 4 | 35 | 200 |
| 4 | 39 | 10 |
| 4 | 40 | 12 |
| 4 | 48 | 10 |

$\overline{\mathbf{AGGR}}^{\mathbf{x,z}}_{\mathbf{f,u}}(\mathbf{S_3})$

| X | Z | SUM |
|---|---|---|
| 1 | 12 | 200 |
| 1 | 18 | 150 |
| 4 | 12 | 100 |
| 4 | 13 | 50 |
| 4 | 20 | 100 |
| 4 | 25 | 150 |
| 4 | 30 | 125 |
| 4 | 33 | 50 |
| 2 | 10 | 30 |
| 2 | 15 | 40 |
| 2 | 21 | 100 |
| 2 | 25 | 150 |
| 2 | 30 | 200 |

Figure 13: $\overline{AGGR}^{x,z}_{f,u}(S_i)$, $i \in \{1, 2, 3\}$.

**phase 5: local computation of the aggregate function**

In this phase, we use Algorithm 2 to evaluate locally the join operation and aggregate function on each processor $i$ represented by $AGGR^{y,z}_{f,u}\big((R \bowtie S)_i\big)$.

Figure 14 shows an example of applying this algorithm on processor 1. For simplicity of presentation, we will use histograms $\overline{Hist}^{x,y}(R_1)$ and $\overline{AGGR}^{x,z}_{f,u}(S_1)$ having small size.

$\overline{\mathbf{Hist}}^{\mathbf{x,y}}(\mathbf{R_1})$

| $X$ | $Y$ | $Freq$ |
|---|---|---|
| 1 | 40 | 300 |
| 3 | 40 | 150 |
| 3 | 6 | 50 |

$\overline{\mathbf{AGGR}}^{\mathbf{x,z}}_{\mathbf{f,u}}(\mathbf{S_1})$

| X | Z | SUM(U) |
|---|---|---|
| 1 | 2 | 1 |
| 3 | 2 | 120 |
| 3 | 7 | 840 |

$\mathbf{AGGR}^{\mathbf{y,z}}_{\mathbf{f,u}}\big((\mathbf{R} \bowtie \mathbf{S})_\mathbf{1}\big)$

| Y | Z | Freq * SUM(U) |
|---|---|---|
| 40 | 2 | 300 * 1 + 150 * 120 |
| 40 | 7 | 150 * 840 |
| 6 | 2 | 50 * 120 |
| 6 | 7 | 50 * 840 |

Figure 14: applying the join operation and aggregate function

**phase 6: global computation of the aggregate function**

After computing $AGGR^{y,z}_{f,u}\big((R \bowtie S)_i\big)$ on each processor i, we redistribute their tuples using a hash function applied on $y$ and $z$ in order to globally compute the aggregate function.

## References


[1] M. Bamha. 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, 2005.

[2] 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*, London, United Kingdom, 2000. Springer-Verlag.

[3] M. Bamha and G. Hains. An efficient equi-semi-join algorithm for distributed architectures. In *Proceedings of the 5th International Conference on Computational Science (ICCS'2005). 22-25 May, Atlanta, USA*, volume 3515 of *Lecture Notes in Computer Science*, pages 755–763. Springer-Verlag, 2005.

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

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

[7] S. Christodoulakis. Implications of certain assumptions in database performance evaluation. *ACM Transactions on Database Systems*, 9(2):163–186, 1984.

[8] 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.

[9] 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.

[10] Ashish Gupta, Venky Harinarayan, and Dallan Quass. 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, 1995. Morgan Kaufmann Publishers Inc.

[11] M. Al Hajj Hassan and M. Bamha. Parallel processing of 'group-by join' queries on Shared Nothing machines. In *Proceedings of the International Conference on Software and Data Technologies (ICSOFT'06)*, volume 1, pages 301–307, Setubal, Portugal, 11-14 September 2006. INSTICC press.

[12] M. Al Hajj Hassan and M. Bamha. An optimal evaluation of groupby-join queries in distributed architectures. In *Proceedings of the third International Conference on Web Information Systems and Technologies (WEBIST 2007)*, volume IT, pages 246–252, Barcelona, Spain, 3 - 6 March 2007. INSTICC press.

[13] Chengkai Li, Kevin Chen-Chuan Chang, and Ihab F. Ilyas. Efficient processing of ad-hoc top-k aggregate queries in olap. Technical report, UIUCDCS-R-2005-2596, Department of Computer Science, UIUC, June 2005.

[14] H. Lu and K.-L. Tan. Dynamic and load-balanced task-oriented database query processing in parallel systems. In *Proceedings of Advances in Database Technology (EDBT '92)*, volume 580 of *LNCS*, pages 357–372, Berlin, Germany, 1992. Springer.

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

[16] Donovan A. Schneider and David J. 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 Management of data, Portland, Oregon, United States, May 1989*, pages 110–121, New York, NY, USA, 1989. ACM Press.

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

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

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

[20] Konrad Stocker, Donald Kossmann, Reinhard Braumandl, and Alfons Kemper. 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, 2001.

[21] D. Taniar, Y. Jiang, K.H. Liu, and C.H.C. Leung. 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, 2000.

[22] David Taniar and J. Wenny Rahayu. 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, 2001.

[23] David Taniar, Rebecca Boon-Noi Tan, C. H. C. Leung, and K. H. Liu. Performance analysis of "groupby-after-join" query processing in parallel database systems. *Inf. Sci. Inf. Comput. Sci.*, 168(1-4):25–50, 2004.

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

[25] Weipeng P. Yan and Per-Åke Larson. Performing group-by before join. In *Proceedings of the Tenth International Conference on Data Engineering*, pages 89–100, Washington, DC, USA, 1994. IEEE Computer Society.

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