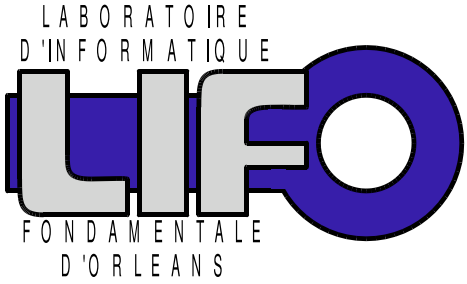

4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE
http://www.univ-orleans.fr/lifo

Rapport de Recherche

Scalability and Optimisation of GroupBy-Joins in MapReduce

Mostafa BAMHA
LIFO, Université d'Orléans
M. Al Hajj Hassan
Lebanese International University, Beirut.

Rapport n^o **RR-2015-01**

Scalability and Optimisation of GroupBy-Joins in MapReduce

M. Al Hajj Hassan[1] and Mostafa Bamha[2]

[1] Lebanese International University, Beirut, Lebanon
mohamad.hajjhassan01@liu.edu.lb
[2] Université Orléans, INSA Centre Val de Loire, LIFO EA 4022, France
Mostafa.Bamha@univ-orleans.fr

Abstract

For over a decade, MapReduce has become the leading programming model for parallel and massive processing of large volumes of data. This has been driven by the development of many engines and frameworks such as Spark, Pig, Hive, facilitating data analysis on large-scale systems. However, these frameworks still remain vulnerable to communication costs, data skew and tasks imbalance problems. This can have a devastating effect on the performance and on the scalability of these systems, more particularly when treating GroupBy-Join queries of large datasets.

In this paper, we present a new *GroupBy-Join* algorithm allowing to reduce communication costs considerably while avoiding data skew effects. A cost analysis of this algorithm shows that our approach is insensitive to data skew and ensures perfect balancing properties during all stages of GroupBy-Join computation even for highly skewed data. These performances have been confirmed by a series of experimentations.

Keywords: Join and GroupBy-join operations, Data skew, MapReduce programming model, Distributed file systems, Hadoop framework, Apache Pig Latin.

1 Introduction

Business intelligence and large-scale data analysis have been recently the object of increased research activity using MapReduce model and especially in the evaluation of complex queries involving GroupBy-Joins using hash based approach [1, 9, 13]. GroupBy-joins still suffer from the effect of high redistribution cost, disk I/O and task imbalance in the presence of skewed data in large scale systems.

GroupBy-Join queries are queries involving join and group-by operations in addition to aggregate functions. In these queries, aggregate functions allow us to obtain a summary data for each group of tuples based on a designated grouping. We can distinguish two types of GroupBy-Join queries as illustrated in the following table. The difference between queries $\mathcal{Q}_1$

Query $\mathcal{Q}_1$		Query $\mathcal{Q}_2$	
SELECT	$R.x, R.y, S.z, f(S.u)$	SELECT	$R.y, S.z, f(S.u)$
FROM	R, S	FROM	R, S
WHERE	$R.x = S.x$	WHERE	$R.x = S.x$
GROUP BY	$R.x, R.y, S.z$	GROUP BY	$R.y, S.z$

Table 1: Different types of "GroupBy-Join" queries.

and $\mathcal{Q}_2$ resides in the `GroupBy` and `Join` attributes. In query $\mathcal{Q}_1$, the join attribute x is part of the `GroupBy` attributes which is not the case in query $\mathcal{Q}_2$. This difference plays an important role in processing the queries especially on parallel and distributed systems.

In traditional algorithms that treat such queries, join operation is performed in the first step and then the group-by operation [3, 14]. But the response time of these queries may be significantly reduced if the group-by operation and aggregate function are performed before the join [3]. This helps in reducing the size of the relations to be joined. In addition, redistribution of tuples of both relations is necessary in join evaluation on parallel and distributed systems. Thus, reducing the size of relations to be joined helps in reducing the communication cost and by consequence in reducing the execution time of queries in parallel and distributed systems. Several optimization techniques were introduced in the literature in order to generate the query execution plan with the lowest processing costs [3,6,14,15]. Their aim is to study the necessary and sufficient conditions that must be satisfied by the relational query in order to be able to push the GroupBy past join operation and to find when this transformation helps in decreasing the execution time. In general, when the join attributes are part of the group-by attributes, as in query $\mathcal{Q}_1$, it is preferable to evaluate the group-by and aggregate function first and then the join operation [7, 8, 12]. In the contrary, group-by cannot be applied before the join in query $\mathcal{Q}_2$ because the join attribute x is not part of the group-by attributes [10, 11].

We have recently proposed in [9], *MRFA-Join* algorithm, a MapReduce based algorithm for evaluating join operations on DFS. *MRFA-Join* algorithm is a scalable and skew-insensitive join algorithm based on distributed histograms and on a randomised key redistribution approach while guaranteeing perfect balancing during all stages of join computation. *MRFA-Join* algorithm, or other MapReduce hash based join algorithms presented in the literature [1,16], could be easily extended to evaluate *GroupBy-Join* queries by adding a final job that redistributes the join result based on the values of the select attributes ($(R.x, R.y, S.z)$ for query $\mathcal{Q}_1$ and $(R.y, S.z)$ for query $\mathcal{Q}_2$). However, this does not allow us to benefit from the optimization techniques described above.

In this paper, we introduce a new *GroupBy-Join* algorithm called `MRFAG-Join` (MapReduce Frequency Adaptive Groupby-Join) based on distributed histograms to get detailed information about data distribution. Information provided by distributed histograms in both *MRFA-Join* and MRFAG-join algorithms allow us to balance load processing among processors nodes due to the fact that all the generated join tasks and buffered data never exceed a user defined size, using threshold frequencies, while reducing communication costs to only relevant data. Moreover in `MRFAG-Join`, we partially apply the group-by operation and aggregate function before evaluating the join. In addition, we do not fully materialize the join result. This helps us to reduce the communication and disk input/output costs to a minimum while preserving the efficiency and the scalability of the algorithm even for highly skewed data. We recall that all existing MapReduce GroupBy-Join algorithms presented in the literature are derived from parallel hashing approaches which make them very sensitive to data skew.

2 The MapReduce Programming Model

MapReduce [4] is a simple yet powerful framework for implementing distributed applications without having extensive prior knowledge of issues related to data redistribution, task allocation or fault tolerance in large scale distributed systems.

Google's MapReduce programming model presented in [4] is based on two functions: **map** and **reduce**, that the programmer is supposed to provide to the framework. These two functions should have the following signatures:

map: $(k_1, v_1) \longrightarrow list(k_2, v_2)$,
reduce: $(k_2, list(v_2)) \longrightarrow list(v_3)$.

The user must write the **map** function that has two input parameters, a key k_1 and an associated value v_1. Its output is a list of intermediate key/value pairs (k_2, v_2). This list is partitioned by the MapReduce framework depending on the values of k_2, where all pairs having the same value of k_2 belong to the same group.

The **reduce** function, that must also be written by the user, has two parameters as input: an intermediate key k_2 and a list of intermediate values $list(v_2)$ associated with k_2. It applies the user defined merge logic on $list(v_2)$ and outputs a list of values $list(v_3)$. In this paper, we

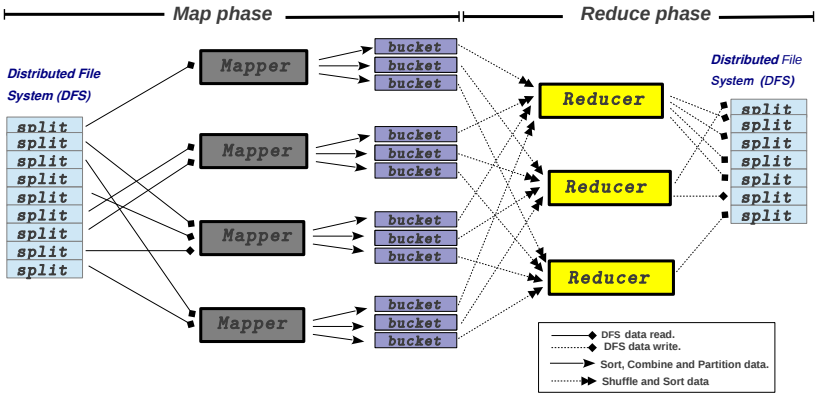

Figure 1: Map-reduce framework.

used an open source version of MapReduce called Hadoop developed by "The Apache Software Foundation". Hadoop framework includes a distributed file system called HDFS[1] designed to store very large files with streaming data access patterns.

For efficiency reasons, in Hadoop MapReduce framework, users may also specify a "Combine function", to reduce the amount of data transmitted from Mappers to Reducers during *shuffle* phase (see fig 1). The "Combine function" is like a local reduce applied (at map worker) before storing or sending intermediate results to the reducers. The signature of **combine** function is:

$$\textbf{combine:} \qquad (k_2, list(v_2)) \longrightarrow (k_2, list(v_3))\,.$$

To cover a large range of applications needs in term of computation and data redistribution, in Hadoop framework, the user can optionally implement two additional functions : **init**() and **close**() called before and after each map or reduce task. The user can also specify a "partition function" to send each key k_2 generated in map phase to a specific reducer destination. The reducer destination may be computed using only a part of the input key k_2. The signature of the **partition** function is:

$$\textbf{partition:} \qquad k_2 \longrightarrow Integer\,,$$

where the output of **partition** should be a positive number strictly smaller than the number of reducers. Hadoop's default **partition** function is based on "hashing" the whole input key k_2.

[1]HDFS: Hadoop Distributed File System.

3 MRFAG-Join : a solution for data skew for GroupBy joins using MapReduce model

In this section, we describe the implementation of MRFAG-Join, to evaluate `GroupBy-join` query $\mathcal{Q}_1$, defined in section 1, using Hadoop MapReduce framework as it is, without any modification. Therefore, the support for fault tolerance and load balancing in MapReduce and Distributed File System are preserved if possible: the inherent load imbalance due to repeated values must be handled efficiently by the join algorithm and not by the MapReduce framework.

In query $\mathcal{Q}_1$, "x" refers to join attribute, attributes "y" and "z" are called `Select attributes` whereas attribute "u" refers to `Aggregate` attribute. We assume that input relations (or datasets) R and S are divided into blocks (splits) of data. These splits are stored in Hadoop Distributed File System (DFS). These splits are also replicated on several nodes for reliability issues. Throughout this paper, for a relation $T \in \{R, S\}$, we use the following notations:

- $|T|$: number of pages (or blocks of data) forming T,
- $||T||$: number of tuples (or records) in relation T,
- $\overline{T}$: the restriction (a fragment) of relation T which contains tuples which appear in the final GroupBy-join result. $||\overline{T}||$ is, in general, very small compared to $||T||$,
- T_i^{map} : the split(s) of relation T affected to mapper i,
- T_i^{red}: the split(s) of relation T affected to reducer i, after a shuffle step,
- $\overline{T_i}$: the split(s) of relation $\overline{T}$ affected to mapper i,
- $||T_i||$: number of tuples in split T_i,
- $Hist^x(T_i^{map})$: Mapper's local histogram of T_i^{map}, i.e. the list of pairs (k, n_k) where k is a value of attribute "x" and n_k its corresponding frequency in relation T_i^{map} on mapper i,
- $Hist_i^x(T)$: the fragment of global histogram of relation T of attribute "x" on reducer i,
- $Hist_i^x(T)(v_x)$ is the global frequency n_{v_x} of value v_x of attribute "x" in relation T,
- $Hist^{x,y}(T_i)$: is the local histogram of relation T_i with respect to both "x" and "y": the list of pairs ($(v_x$,$v_y)$,$n_{x,y}$) where v_x and v_y are respectively values of attributes "x" and "y" and $n_{x,y}$ its corresponding frequency in relation T_i.
- $Hist_i^{x,y}(T)$: is the fragment of $Hist^{x,y}(T)$ affected to reducer i,
- $AGGR_{f(u)}^w(T_i)$: is the result of applying the aggregate function f on the values of the aggregate attribute u on every group of tuples of T_i having identical values of the group-by attribute w. $AGGR_{f(u)}^w(T_i)$ is formed of a list of tuples having the form (v, f_v) where f_v is the result of applying the aggregate function on the group of tuples, in T_i, having value v for the attribute w (w may be formed of more than one attribute),
- $AGGR_{f(u),i}^w(T)$: is the fragment of $AGGR_{f(u)}^w(T)$ affected to reducer i,
- $HistIndex(R \bowtie S)$: join attribute values that appear in both R and S and their corresponding three parameters: *Frequency_index*, *Nb_buckets1* and *Nb_buckets2* used in communication templates,
- $c_{r/w}$: read/write cost of a page of data from/to distributed file system (DFS),
- c_{comm}: communication cost per page of data,
- t_s^i: time to perform a simple search in a Hashtable on node i,
- t_h^i: time to add an entry to a Hashtable on node i,
- $NB\_mappers$: number of job mapper nodes,
- $NB\_reducers$: number of job reducer nodes.

We will describe MRFAG-Join algorithm while giving a cost analysis for each computation phase. Join computation in MRFAG-Join proceeds in three MapReduce jobs:

a. the first map-reduce job is performed to compute global distributed histogram and to create randomized communication templates to redistribute only relevant data while avoiding the effect of data skew,

b. the second one, is used to compute partial aggregation of relevant data by using communication templates carried out in the previous job,

c. the third job, is used to redistribute relevant partial Aggregated data by using communication templates carried out in the first job and then compute final GroupBy-join result.

The O(...) notation only hides small constant factors: they only depend on program's implementation but neither on data nor on machine parameters. Data redistribution in MRFAG-Join algorithm is the basis for efficient and scalable join processing while avoiding the effect of data skew in all the stages of `GroupBy-join` computation. MRFAG-Join algorithm (see Appendix Algorithm 1) proceeds in 6 steps:

a.1: Map phase to generate tagged "local histograms" for input relations:
In this step, each mapper i reads its assigned data splits (blocks) of relations R and S from distributed file system (DFS) and emits a couple (<K,tag>,1) for each record in R_i^{map} (resp. S_i^{map}) where K is a value of join attribute "x" and *tag* represents input relation tag. The cost of this step is :

$$Time(a.1.1) = O\big(\max_{i=1}^{NB\_mappers} c_{r/w} * (|R_i^{map}| + |S_i^{map}|) + \max_{i=1}^{NB\_mappers}(||R_i^{map}|| + ||S_i^{map}||)\big).$$

Emitted couples (<K,tag>,1) are then combined and partitioned using a user defined partitioning function by hashing only key part K and not the whole mapper tagged key <K,tag>. The result of combine phase is then sent to destination reducers in the shuffle phase of the following reduce step. The cost of this step is at most :

$$Time(a.1.2) = O\Bigg(\max_{i=1}^{NB\_mappers}\big(||Hist^x(R_i^{map})|| * \log||Hist^x(R_i^{map})|| + ||Hist^x(S_i^{map})|| * \log||Hist^x(S_i^{map})||\big) + c_{comm} * (|Hist^x(R_i^{map})| + |Hist^x(S_i^{map})|)\Bigg).$$

And the global cost of this step is: $Time(\mathbf{a.1}) = Time(a.1.1) + Time(a.1.2)$.

We recall that, in this step, only local histograms $Hist^x(R_i^{map})$ and $Hist^x(S_i^{map})$ are sorted and transmitted across the network and the sizes of these histograms are very small compared to the size of input relations R_i^{map} and S_i^{map} owing to the fact that, for a relation T, $Hist^x(T)$ contains only distinct entries of the form (k, n_k) where k is a value of join attribute "x" and n_k its corresponding frequency.

a.2: Reduce phase to create join result global histogram index and randomized communication templates for relevant data:
At the end of shuffle phase, each reducer i will receive a fragment of $Hist_i^x(R)$ (resp. $Hist_i^x(S)$) obtained through hashing of distinct values of $Hist^x(R_j^{map})$ (resp. $Hist^x(S_j^{map})$) of each mapper j. Received fragments of $Hist_i^x(R)$ and $Hist_i^x(S)$ are then merged to compute global histogram $HistIndex_i(R \bowtie S)$ on each reducer i. $HistIndex(R \bowtie S)$ is used to compute randomized communication templates for only records associated to relevant join attribute values (i.e. values which will effectively be present in the final *GroupBy-Join* result).

In this step, each reducer i, computes the global frequencies for join attribute values which are present in both left and right relations and emits, for each join attribute K, an entry of the form : (K,<Frequency_index(K),Nb_buckets1(K),Nb_buckets2(K)>) where:

- $Frequency\_index(K) \in \{0, 1, 2\}$ will allow us to decide if, for a given relevant join attribute value K, the frequencies of tuples in relations R and S, having the value K, are greater (resp. smaller) than a defined threshold frequency f_0. It also permits us to choose dynamically the probe and the build relation for each value K of the join attribute. This choice reduces the global redistribution cost to a minimum.
  For a given join attribute value $K \in HistIndex_i(R \bowtie S)$,

$$\begin{cases} \text{Frequency\_index(K)=0} & \text{If } Hist_i^x(R)(K) < f_0 \text{ and } Hist_i^x(S)(K) < f_0 \\ & \text{(i.e. values associated to low frequencies in both relations),} \\ \text{Frequency\_index(K)=1} & \text{If } Hist_i^x(R)(K) \geq f_0 \text{ and } Hist_i^x(R)(K) \geq Hist_i^x(S)(K) \\ & \text{(i.e. Frequency in relation R is higher than those of S),} \\ \text{Frequency\_index(K)=2} & \text{If } Hist_i^x(S)(K) \geq f_0 \text{ and } Hist_i^x(S)(K) > Hist_i^x(R)(K) \\ & \text{(i.e. Frequency in relation S is higher than those of R).} \end{cases}$$

- Nb_buckets1(K): is the number of buckets used to partition records of relation associated to the highest frequency for join attribute value K,
- Nb_buckets2(K): is the number of buckets used to partition records of relation associated to the lowest frequency for join attribute value K.

For a join attribute value K, the number of buckets $Nb\_buckets1(K)$ and $Nb\_buckets2(K)$ are generated in a manner that each bucket will fit in reducer's memory. This makes the algorithm insensitive to the effect of data skew even for highly skewed input relations.

Figure 2 gives an example of communication templates used to partition data for $HistIndex$ entry (K,<Frequency_index(K),Nb_buckets1(K),Nb_buckets2(K)>) corresponding to a join attribute K associated to a high frequency, into small buckets. In this example, data associated to relation corresponding to Tag_1 is partitioned into 5 buckets (i.e. $Nb\_buckets1(K) = 5$) whereas those of relation corresponding to Tag_2 is partitioned into 3 buckets (i.e. $Nb\_buckets2(K) = 3$). For these buckets, appropriate map keys are generated so that all records in each bucket of relation associated to Tag_1 are forwarded to the same reducer holding all the buckets of relation associated to Tag_2. This partitioning guarantees that join tasks, are generated in a manner that the input data for each join task will fit in the memory of processing node and never exceed a user defined size, even for highly skewed data [9].

Using $HistIndex$ information, each reducer i, has local knowledge of how relevant records of input relations will be redistributed in the next map phase. The global cost of this step is at most: $Time(\mathbf{a.2}) = O\big(\max_{i=1}^{NB\_reducers}(||Hist_i^x(R)|| + ||Hist_i^x(S)||)\big)$.
Note that, $HistIndex(R \bowtie S) \equiv \cup_i(Hist_i^x(R) \cap Hist_i^x(S))$ and $||HistIndex(R \bowtie S)||$ is very small compared to $||Hist^x(R)||$ and $||Hist^x(S)||$.

To guarantee a perfect balancing of the load among processing nodes, communication templates are carried out jointly by all reducers (and not by a coordinator node) for only join attribute values which are present in *GroupBy-Join* result : Each reducer deals with the redistribution of the data associated to a subset of relevant join attribute values.

b.1: Map phase to create a local hashtable and to generate relevant partial aggregated data :
In this step, each mapper i reads join result global histogram index, $HistIndex$, to create a local hashtable in time: $Time(b.1.1) = O(\max_{i=1}^{NB\_mappers} t_h^i * ||HistIndex(R \bowtie S)||)$.
Once local hashtable is created on each mapper, input relations are then read from DFS, and for each relevant record (record having join key present in the local hashtable) is either discarded (if record's join key is not present in the local hashtable) or routed to a designated random

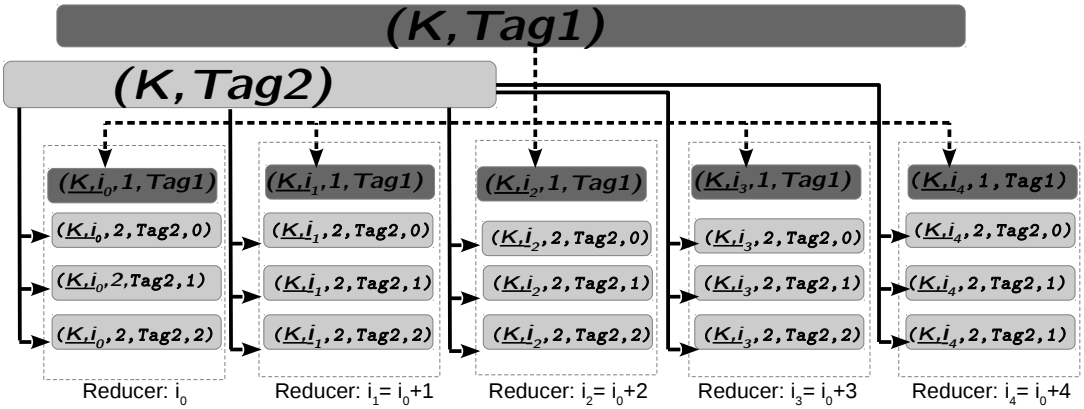

Figure 2: Generated buckets associated to a join key K corresponding to a high frequency where records from relation associated to Tag_1 (i.e relation having the highest frequency) are partitioned into five buckets and those of relation associated to Tag_2 are partitioned into three buckets.

reducer destination using communication templates computed in step a.2 (Map phase details are described in Algorithm 5 of appendix). The cost of this step is:

$$Time(b.1.2) = O\Bigg(\max_{i=1}^{NB\_mappers}(c_{r/w} * (|R_i^{map}| + |S_i^{map}|) + t_s^i * (||R_i^{map}|| + ||S_i^{map}||) + ||Hist^{x,y}(\overline{R}_i^{map})|| * \log ||Hist^{x,y}(\overline{R}_i^{map})|| + ||AGGR_{f(u)}^{x,z}(\overline{S}_i^{map})|| * \log ||AGGR_{f(u)}^{x,z}(\overline{S}_i^{map})|| + c_{comm} * (|Hist^{x,y}(\overline{R}_i^{map})| + |AGGR_{f(u)}^{x,z}(\overline{S}_i^{map})|)) \Bigg).$$

The term $c_{r/w} * (|R_i^{map}| + |S_i^{map}|)$ is time to read input relations from DFS on each mapper i, the term $t_s^i * (||R_i^{map}|| + ||S_i^{map}||)$ is the time to perform a hashtable search for each mapper's input record, $||Hist^{x,y}(\overline{R}_i^{map})|| * \log ||Hist^{x,y}(\overline{R}_i^{map})|| + ||AGGR_{f(u)}^{x,z}(\overline{S}_i^{map})|| * \log ||AGGR_{f(u)}^{x,z}(\overline{S}_i^{map})||$ is time to sort relevant aggregated data on mapper i, whereas the term $c_{comm} * (|Hist^{x,y}(\overline{R}_i^{map})| + |AGGR_{f(u)}^{x,z}(\overline{S}_i^{map})|)$ is time to communicate relevant aggregated data from mappers to reducers. Hence the global cost of this step is: $Time(\mathbf{b.1}) = Time(b.1.1) + Time(b.1.2)$.

b.2: Reduce phase to compute relevant partial aggregated data, $Hist_i^{x,y}(\overline{R})$ and $AGGR_{f(u),i}^{x,z}(\overline{S})$:
At the end of step b.1, each reducer i receives a fragment $Hist_i^{x,y}(\overline{R})$ (resp. $AGGR_{f(u),i}^{x,z}(\overline{S})$) obtained through a hashing of $Hist^{x,y}(\overline{R}_j^{map})$ (resp. $AGGR_{f(u)}^{x,z}(\overline{S}_j^{map})$) of each mapper j and then a local merge of received data. This partial aggregated data is then written to DFS. This reduce phase is described in detail in Algorithm 8 of appendix. The cost of this step is:

$$Time(\mathbf{b.2}) = O(\max_{i=1}^{NB\_reducers}(||Hist_i^{x,y}(\overline{R})|| + ||AGGR_{f(u),i}^{x,z}(\overline{S})||) + c_{r/w} * |Hist_i^{x,y}(\overline{R})| + |AGGR_{f(u),i}^{x,z}(\overline{S})|)).$$

c.1: Map phase to create a local hashtable and to redistribute relevant aggregated data using randomized communication templates:
In this step, each mapper i reads join result global histogram index, $HistIndex$, to create a local

hashtable in time: $Time(c.1.1) = O(\max_{i=1}^{NB\_mappers} t_h^i * ||HistIndex(R \bowtie S)||)$.
Once local hashtable is created on each mapper, input relations are then read from DFS, and each record is either discarded (if record's join key is not present in the local hashtable) or routed to a designated random reducer destination using communication templates computed in step a.2 (Map phase is detailed in Algorithm 9 of Appendix). The cost of this step is:

$$Time(c.1.2) = O\Bigg(\max_{i=1}^{NB\_mappers} (c_{r/w} * (|Hist_i^{x,y}(\overline{R})| + |AGGR_{f(u),i}^{x,z}(\overline{S})|) + t_s^i * ||Hist_i^{x,y}(\overline{R})|| + t_s^i * ||AGGR_{f(u),i}^{x,z}(\overline{S})|| + ||Hist_i^{x,y}(\overline{R})|| * \log ||Hist_i^{x,y}(\overline{R})|| + ||AGGR_{f(u),i}^{x,z}(\overline{S})|| * \log ||AGGR_{f(u),i}^{x,z}(\overline{S})|| + c_{comm} * (|Hist_i^{x,y}(\overline{R})| + |AGGR_{f(u),i}^{x,z}(\overline{S})|)) \Bigg).$$

The term $c_{r/w} * (|Hist_i^{x,y}(\overline{R})| + |AGGR_{f(u),i}^{x,z}(\overline{S})|)$ is time to read input relations from DFS on each mapper i, the term $t_s^i * (||Hist_i^{x,y}(\overline{R})|| + ||AGGR_{f(u),i}^{x,z}(\overline{S})||)$ is the time to perform a hashtable search for each input record, $||Hist_i^{x,y}(\overline{R})|| * \log ||Hist_i^{x,y}(\overline{R})|| + ||AGGR_{f(u),i}^{x,z}(\overline{S})|| * \log ||AGGR_{f(u),i}^{x,z}(\overline{S})||$ is time to sort relevant data on mapper i, whereas the term $c_{comm} * (|Hist_i^{x,y}(\overline{R})| + |AGGR_{f(u),i}^{x,z}(\overline{S})|))$ is time to communicate relevant data from mappers to reducers, using our communication templates described in step $a.2$. Hence the global cost of this step is:

$$Time(\mathbf{c.1}) = Time(c.1.1) + Time(c.1.2).$$

We recall that, in this step, only relevant data is emitted by mappers (which reduces communication cost in the shuffle step to a minimum) and records associated to high frequencies (those having a large effect on data skew) are redistributed according to an efficient dynamic partition/replicate schema to balance load among reducers and avoid the effect of data skew. However records associated to low frequencies (these records have no effect on data skew) are redistributed using hashing functions.

c.2: Reduce phase to compute join result:
At the end of step b.1, each reducer i receives a fragment $Hist_i^{x,y}(\overline{R})$ (resp. $AGGR_{f(u),i}^{x,z}(\overline{S})$) obtained through randomized hashing of $\overline{R}_j^{map}$ (resp. $\overline{S}_j^{map}$) of each mapper j and performs a local GroupBy-join of received data. This reduce phase is described in detail in Algorithm 11 of Appendix. The cost of this step is:

$$Time(\mathbf{c.2}) = O(\max_{i=1}^{NB\_reducers} (||Hist_i^{x,y}(\overline{R})|| + ||AGGR_{f(u),i}^{x,z}(\overline{S})|| + c_{r/w} * |Hist_i^{x,y}(\overline{R}) \bowtie AGGR_{f(u),i}^{x,z}(\overline{S})|)).$$

Figure 3 shows how the frequency in $Hist^{x,y}(\overline{R})$ and the value of aggregate function in $AGGR_{f(u)}^{x,z}(\overline{S})$ are used to generate final `GroupBy-Join` result depending on the used aggregate function f. The final `GroupBy-Join` is computed in two steps. In the first step, a partial local computation of the aggregate function is performed on S entries by the combiners (step b.1 of Algorithm 1). The computation varies depending on the aggregate function f as follows:

- MIN function: the minimum value between the list of values emitted by the mappers for each key is computed.
- MAX function: the maximum value between the list of values emitted by the mappers for each key is computed.
- COUNT function: the number of emitted values by the mappers for each key is computed.
- SUM function: the sum of emitted values by the mappers for each key is computed.
- AVG function: the sum and the number of emitted values by the mappers are computed for each key. The result is the computed sum divided by the computed count value.

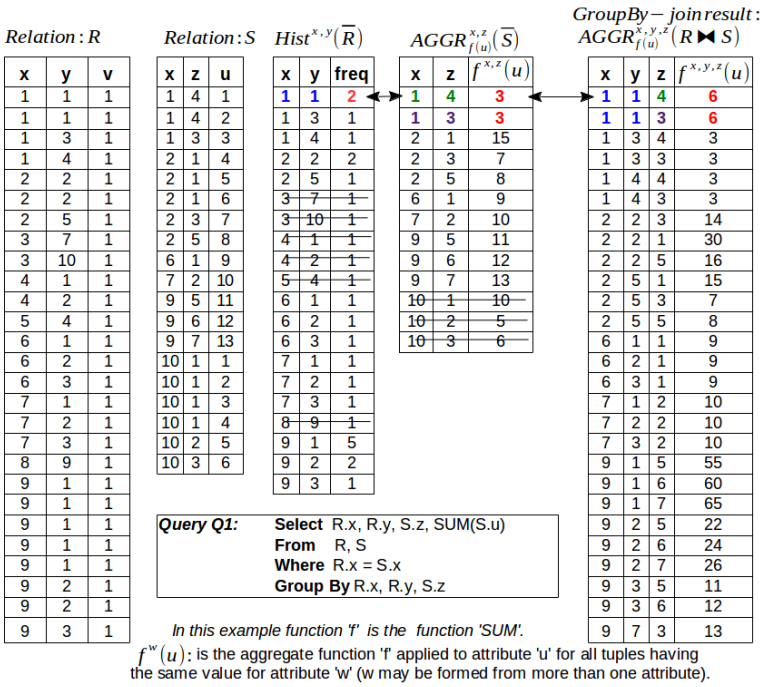

Figure 3: Final GroupBy-Join computation using $Hist^{x,y}(\overline{R})$ and $AGGR^{x,z}_{f(u)}(\overline{S})$.

In the second step, reducers apply a similar treatment on received data to compute the global result (step c.2 of Algorithm 1). However, for COUNT, SUM and AVG functions the frequencies for values (x, y) in R should be multiplied by the result of the aggregate function on S.

The global cost of MRFAG-Join is therefore the sum of the above six steps:

$$Time_{MRFAG-Join} = Time(\mathbf{a.1}) + Time(\mathbf{a.2}) + Time(\mathbf{b.1}) + Time(\mathbf{b.2}) + Time(\mathbf{c.1}) + Time(\mathbf{c.2}).$$

Using hashing technique, the computation of query $\mathcal{Q}_1$ requires at least the following lower bound:

$$bound_{inf} = \Omega\Big(\max_{i=1}^{NB\_mappers} \big((c_{r/w} + c_{comm}) * (|R_i^{map}| + |S_i^{map}|) + ||R_i^{map}|| * \log ||R_i^{map}|| + ||S_i^{map}|| * \log ||S_i^{map}||\big) + \max_{i=1}^{NB\_reducers} \big(||R_i^{red}|| + ||S_i^{red}|| + c_{r/w} * |AGGR^{x,y,z}_{f(u)}(R_i^{red} \bowtie S_i^{red})|\big)\Big),$$

where $c_{r/w} * (|R_i^{map}| + |S_i^{map}|)$ is the cost of reading input relations from DFS on node i. The term $||R_i^{map}|| * \log ||R_i^{map}|| + ||S_i^{map}|| * \log ||S_i^{map}||$ represents the cost to sort input relations records in map phase. The term $c_{comm} * (|R_i^{map}| + |S_i^{map}|)$ represents the cost to communicate data from mappers to reducers, the term $||R_i^{red}|| + ||S_i^{red}||$ is time to scan input relations on reducer i. (Note that for relation $T \in \{R, S\}$, T_i^{red} is the part of T held by reducer i by using hashing function to redistribute data from mappers to reducers) and $c_{r/w} * |AGGR^{x,y,z}_{f(u)}(R_i^{red} \bowtie S_i^{red})|$ represents the cost to store reducer's i `GroupBy-Join` result on the DFS for query $\mathcal{Q}_1$.

MRFAG-Join algorithm has asymptotic optimal complexity when: $||HistIndex(R \bowtie S)|| \leq$

$$\max\left(\max_{i=1}^{NB\_mappers}(||R_i^{map}||*\log||R_i^{map}||, |S_i^{map}||*\log||S_i^{map}||), \max_{i=1}^{NB\_reducers}||AGGR_{f(u)}^{x,y,z}(R_i^{red} \bowtie S_i^{red})||\right), \quad (1)$$

this is due to the fact that, all other terms in $Time_{MRFAG-Join}$ are bounded by those of $bound_{inf}$. Inequality 1 holds, in general, since $HistIndex(R \bowtie S)$ contains only distinct values that appear in both relations R and S.

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

- *a bad choice of used hash function.* 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 [2],
- *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. There is no way for a clever hash function to avoid load imbalance that results from these repeated values [5]. But this case cannot arise here owing to the fact that histograms contain only distinct values of the join attribute and the hashing functions we use are always applied to histograms or applied to aggregated data.

4 Experiments

To evaluate the performance of `MRFAG-Join` algorithm presented in this paper, for query $\mathcal{Q}_1$, we compared our algorithm to the best known solution using PigLatin a high-level language for analyzing large data sets based on Apache Pig plateform which generates optimized MapReduce programs. Pig includes routines to handle the effects of data skew in join operations.

We ran a large series of experiments on the Grid'5000 testbed where 50 nodes were randomly selected from three clusters of Grid'5000 Sophia's site. Nodes characteristics are described in Table 2. Setting up a Hadoop cluster consisted of deploying each centralized entity (namenode and jobtracker) on a dedicated machine and co-deploying datanodes and tasktrackers on the rest of the nodes. Typically, we used a separate machine as a Hadoop client to manage job submissions. Data replication parameter was fixed to three in Hadoop Distributed File System (HDFS) configuration file.

To study the effect of data skew on performance, join attribute values in the generated data

Table 2: Grid'5000 - Sophia's site computing resource characteristics

Cluster ID	Number of nodes	CPU	CPUs per node	Cores per CPU	Memory (GB)	Disk Storage
1	56	AMD@2.2GHz	2	2	3GB RAM	135GB
2	50	AMD@2.6GHz	2	2	3GB RAM	232GB
3	45	Intel@2.26GHz	2	4	31GB RAM	557GB

have been chosen to follow a Zipf distribution [17] as it is the case in most database tests: Zipf factor was varied from 0 (for a uniform data distribution) to 1.0 (for a highly skewed data). Input relations sizes were fixed to 1 Billion records for the right relation (∼100GB of data) and 500M of records for the left relation ∼50GB of data) and the GroupBy-Join result varying from

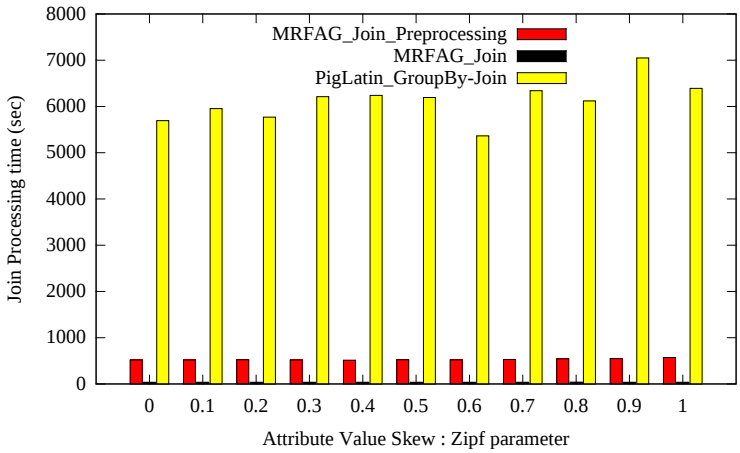

Figure 4: Data skew effect on Hadoop GroupBy-join processing time

approximately 20M to 50M records (corresponding respectively to about 400MB and 1GB of aggregated output data).
We noticed in all the tests and also those presented in Figure 4, that our `MRFAG-Join` (including histogram's and aggregate data preprocessing) algorithm outperforms `PigLatin_GroupBy_Join` execution even for low or moderated skew ($\sim$ 10x faster than `PigLatin_GroupBy_Join`). We recall that our algorithm requires the scan of input data twice. The first scan is performed for histograms processing and the second one to generate relevant partial aggregated data.
We can see, in Figure 4, that `MRFAG-Join` time is relatively very small compared to `MRFAG-Join` preprocessing time this is explained by the fact that `MRFAG_Join_Preprocessing` operates on whole input data (to generate distributed histograms or relevant aggregated data) whereas `MRFAG_Join` operates on relevant aggregated data which is very small compared to the size of input relations.

The cost analysis and tests performed showed that the overhead related to histogram processing is compensated by the gain in GroupBy-join processing since only relevant data (that appears in the final GroupBy-Join result) is emitted by mappers which reduce considerably the amount of data transmitted over the network in shuffle phase (see Figure 5). Moreover, in `PigLatin_GroupBy_Join` implementation all records, emitted by the mappers, having the same value for join attribute are sent and processed by the same reducer which makes the algorithm very sensitive to data skew and limits its scalability. This cannot occur in MRFAG-Join owing to the fact that attribute values associated to high frequencies are forwarded to distinct reducers using randomised join attribute keys and not by a simple hashing of record's join key.

5 Conclusion and future work

In this paper, we have presented MRFAG-Join (*MapReduce Frequency Adaptive GroupBy-Join*) algorithm using MapReduce model based on distributed histograms and a randomised key redistribution approach. This algorithm achieves several enhancements compared to hash based solutions suggested in the literature by reducing communication costs to only relevant or aggregated data while guaranteeing perfect balancing properties even for highly skewed data. The cost analysis and the tests performed on Grid5000 infrastructure showed that the overhead re-

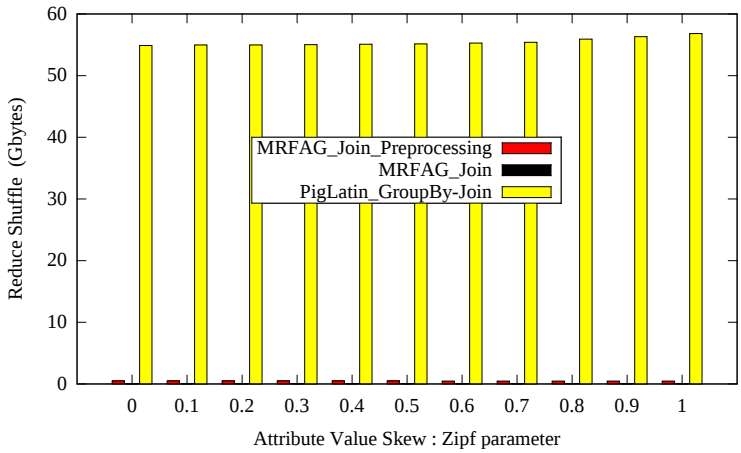

Figure 5: Data skew effect on the amount of data moved across the network during shuffle phase

lated to distributed histogram management remains very small compared to the gain it provides to avoid the effect of load imbalance due to data skew, and to reduce the communication costs in MapReduce'shuffle phase. We recall that, data processing, unnecessary disk I/O and redistribution of the intermediate results can lead to a significant degradation of the performance using 'pure' hash based approaches presented in the literature to perform *GroupBy* joins on large datasets.
Future work will be devoted to extend this algorithm to multi-join and pipelined *GroupBy-join* queries on large scale systems.

Acknowledgements

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see `https://www.grid5000.fr`).

References

[1] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in mapreduce. In *Proceedings of ACM SIGMOD International Conference on Management of data*, pages 975–986, New York, USA, 2010.

[2] J. Lawrence Carter and Mark N. Wegman. Universal Classes of Hash Functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[3] Surajit Chaudhuri and Kyuseok Shim. Including group-by in query optimization. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 354–366, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[4] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150. USENIX Association, 2004.

[5] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical Skew Handling in Parallel Joins. In *VLDB*, pages 27–40, 1992.

[6] 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.

[7] 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.

[8] 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.

[9] M. Al Hajj Hassan, M. Bamha, and F. Loulergue. Handling data-skew effects in join operations using mapreduce. In *International Conference on Computational Science (ICCS)*, pages 145–158. Elsevier, 2014.

[10] Yi Jiang, Kevin H. Liu, and Clement H. C. Leung. Parallel algorithms for queries with aggregate functions in the presence of data skew. In *HiPC '99: Proceedings of the 6th International Conference on High Performance Computing*, pages 207–211, London, UK, 1999. Springer-Verlag.

[11] 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.

[12] David Taniar and Wenny Rahayu. Parallel "groupby-before-join" query processing for high performance parallel/distributed database systems. In *Proceedings of the 20th International Conference on Advanced Information Networking and Applications - Volume 1*, pages 693–700, Washington, USA, 2006. IEEE Computer Society.

[13] Srinivas Vemuri, Maneesh Varshney, Krishna Puttaswamy, and Rui Liu. Execution primitives for scalable joins and aggregations in map reduce. *PVLDB*, 7(13):1462–1473, 2014.

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

[15] Weipeng P. Yan and Per-Åke Larson. Eager aggregation and lazy aggregation. In *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases*, pages 345–357, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[16] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *the ACM SIGMOD international conference on Management of data*, pages 1029–1040, New York, USA, 2007.

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

6 Appendix: Implementation of MRFAG-Join functions

Algorithm 1 MRFAG-join algorithm workflow for query $\mathcal{Q}_1$ /* See Appendix for detailed implementation */

a.1 ▶ **Map Phase** /* To generate "local histograms" $Hist^x(R_i^{map})$ and $Hist^x(S_i^{map})$ */
- ▷ Each mapper i reads its assigned data splits of input relations R_i^{map} and S_i^{map} from DFS
- ▷ Extracts join_key value from input relation's record.
- ▷ Gets a tag to identify source input relation
- ▷ Emits a couple ((join_key,tag), 1)

▶ **Combine Phase**

▷ Each combiner, for each pair (join_key,tag) computes the sum of generated local frequencies associated to the join_key value in each tagged join_key generated in Map phase.

▶ **Partition Phase**

▷ For each emitted tagged join_key computes reducer destination according to only join_key value.

a.2 ▶ **Reducer Phase**: /* To combine shuffle's records and to create Global join histogram index*/
- ▷ Compute the global frequencies for only join_key values present in both R and S.
- ▷ Emit, for each join key, a couple (join_key,(frequency_index,Nb_buckets1, Nb_buckets2)).

/*frequency_index $\in \{0, 1, 2\}$ is used to get detailed information of data distribution in R and S*/

b.1 ▶ **Map Phase** /* To compute $Hist^{x,y}(\overline{R}_i^{map})$ and $AGGR_{f(u)}^{x,z}(\overline{S}_i^{map})$ */
- ▷ Each mapper i reads Global join histogram index from DFS and creates a local HashTable
- ▷ Each mapper i reads its assigned data splits of relations R_i^{map} and S_i^{map} from DFS
- ▷ Extract join_key value from input relation's record.

If (join_key ∈ HashTable) **Then**
- ▷ Extract the groupby attribute y of R (resp. z of S in addition to the aggregate attribute u)
- ▷ Get a tag to identify source input relation
- ▷ Emit a couple ((join_key,groupby_attribute,tag), n) where n is 1 for tuples relation R and the aggregate attribute's value for S.

End If

▶ **Combine Phase**

▷ Each combiner, for each record with key (join_key,groupby_attribute,tag) computes the sum of generated local frequencies for R and applies aggregate function on values of attribute u of S.

▶ **Partition Phase**

▷ For each emitted tagged (join_key, groupby_attribute), computes reducer destination according to join_key and groupby_attribute values.

b.2 ▶ **Reduce Phase**: /* To create $Hist_i^{x,y}(\overline{R})$ and $AGGR_{f(u),i}^{x,z}(\overline{S})$*/

▷ For each key (join_key,groupby_attribute,tag) compute, $freq^{x,y}(R)$, the global sum of generated local frequencies for R and globally compute, $f_u^{x,z}(S)$, the result of the aggregate function for S.

▷ Emit a couple ((join_key,groupby_attribute, tag),$freq^{x,y}(R)$)
(resp. ((join_key,groupby_attribute, tag),$f_u^{x,z}$(S))).

c.1 ▶ **Map Phase**: /* Repartition of $Hist_i^{x,y}(\overline{R})$ and $AGGR_{f(u),i}^{x,z}(\overline{S})$*/
- ▷ Each mapper i reads Global join histogram index from DFS and creates a local Hashtable
- ▷reads its assigned data splits of $Hist_i^{x,y}(\overline{R})$ and $AGGR_{f(u),i}^{x,z}(\overline{S})$ from DFS,
- ▷ Generates randomized communication templates for records in $Hist_i^{x,y}(\overline{R})$ and $AGGR_{f(u),i}^{x,z}(\overline{S})$ according to join key value and its corresponding frequency index in HashTable.
- ▷ Emits relevant randomised tagged records from relations $Hist_i^{x,y}(\overline{R})$ and $AGGR_{f(u),i}^{x,z}(\overline{S})$.

▶ **Partition Phase**:

▷ For each emitted tagged join key, compute reducer destination according to the value of join attribute and random reducer destination generated in Map phase;

c.2 ▶ **Reduce Phase**: /* To generate final query $\mathcal{Q}_1$ result */
- ▷ Combine received entries to create final result, $AGGR_{f(u),i}^{x,y,z}(R \bowtie S)$, on each reducer i.

Algorithm 2 Map function /* To generate local histograms values of join attribute values and tag input relation records */

```
map(K: null, V : a record from a split of either relation R or S) {
        ▷ relation_tag ← get relation tag from current relation split;
        ▷ join_key ← extract the join column from record V of relation R (resp. S);
        ▷ Emit ((join_key,relation_tag), 1);
}
```

Algorithm 3 Combine function /* To compute local frequencies for join_key*/

```
combine(Key K, List List_V) {      /* List_V is the list of values "1" emitted by Mappers */
        ▷ frequency ← sum of frequencies in List_V;
        ▷ Emit (K,frequency);
}
```

Algorithm 4 Reduce function /* To compute $HistIndex(R \bowtie S)$ Global histogram index */

```
void reduce_init(){
  hash_index ← 0;          /* a flag to identify low frequencies records to redistribute using hashing */
  partition_index ← 1;     /* a flag to identify relation's records to partition */
  replicate_index ← 2 ;    /* a flag to identify relation's records to replicate */
  last_inner_key ← "" ;    /* to store the last processed key in inner relation */
  last_inner_frequency=0; /* to store the frequency of the last processed key in inner relation */
  /* THRESHOLD_FREQ: a user defined threshold frequency used for communication templates */
}
reduce(Key K,List List_V) {/* List_V:list of local frequencies of join_key in either R_i^map or S_i^map */
  ▷ join_key ← K.join_key;            /* extracts join key part from input key K */
  ▷ relation_tag ← K.relation_tag;    /* extracts relation tag part from input key K */
  If (relation_tag corresponds to inner relation ) Then
    ▷ last_inner_key ← join_key;
    ▷ last_inner_frequency ← sum of frequencies in List_V;
  Else If (join_key = last_inner_key) Then
      ▷ frequency ← sum of frequencies in List_V ;
      If ((last_inner_frequency<THRESHOLD_FREQ) and  (frequency<THRESHOLD_FREQ) Then
        ▷ Emit (join_key, (hash_index,1,1));
      ElseIf (last_inner_frequency ≥ frequency)
        ▷ Nb_buckets1 ← ⌈last_inner_frequency / THRESHOLD_FREQ⌉ ;
        ▷ Nb_buckets2 ← ⌈frequency / THRESHOLD_FREQ⌉;
        ▷ Emit (join_key, (partition_index,Nb_buckets1,Nb_buckets2));
      Else
        ▷ Nb_buckets1 ← ⌈frequency / THRESHOLD_FREQ⌉;
        ▷ Nb_buckets2 ← ⌈last_inner_frequency / THRESHOLD_FREQ⌉;
        ▷ Emit (join_key, (replicate_index,Nb_buckets1,Nb_buckets2));
      End If;
    End If;
  End If;
}
```

Algorithm 5 Map function /* To compute $Hist^{x,y}(\overline{R}_i^{map})$ and $AGGR_{f(u)}^{x,z}(\overline{S}_i^{map})$ */

```
void map_init(){
   inner_tag ← 1 ;              /* a tag to identify relation R records */
   outer_tag ← 2 ;              /* a tag to identify relation S records */
   hash_index ← 0;              /* a flag to identify hash based records */
   partition_index ← 1;         /* a flag to identify records to partition */
   replicate_index ← 2 ;        /* a flag to identify records to replicate */
   Read HistIndex(R ⋈ S): histogram index from DFS;
   Create a HashTable using join_key value, frequency's index and Nb_buckets of HistIndex(R ⋈ S);
}
map(K: null, V : a record from a split of either relation R or S) {
      ▷ relation_tag ← get relation tag from current relation split;
      ▷ join_key ← extract the join column from record V of relation R;
      If (join_key ∈ HashTable) Then /* To redistribute only relevant records */
        ▷ groupby_attribute ← extract the group-by attribute's value y (resp. z) from record V
        of R (resp. S);
        ▷ aggregate_attribute ← extract the aggregate attribute's value u from record V
        of relation S;
        ▷ Emit ((join_key,groupby_attribute,relation_tag), 1) for records of R;
        ▷ Emit ((join_key,groupby_attribute,relation_tag), aggregate_attribute) for records of S;
      End If
}
```

Algorithm 6 Combine function: /* To locally compute frequency of each (join_key, group_by attribute) for R and partial result of aggregate function for S */

```
combine(Key K,List List_V ) {      /* List_V is the list of values "1" corresponding to frequencies
            in relation R_i or the aggregate attribute value u of S_i emitted by Mappers */
      If (relation_tag represents R) Then
        ▷ v ← sum of frequencies in List_V;
      Else /* relation_tag represents S*/
        ▷ v ← apply the aggregate function on values of List_V ;
      End If
      ▷ Emit (K,v);
}
```

Algorithm 7 Partitioning function /* Returns for each composite key K=(join_key,groupby_attribute,relation_tag) emitted in Map phase, an integer corresponding to destination reducer for the input key K. */

```
int partition(K: input key ){
    ▷ subkey ← K.join_key + " " + K.groupby_attribute;
    ▷ Return (HashCode(subkey) % NB_reducers);
}
```

Algorithm 8 Reduce function: /* To globally compute frequency of each (join_key, group_by attribute) for R and result of aggregate function for S */

combine(Key K,List List_V) { /* List_V is the list of values "1" corresponding to frequencies in relation R_i or the aggregate attribute value u of S_i emitted by Mappers */

If (relation_tag represents R) **Then**

▷ $v \leftarrow$ sum of frequencies in List_V;

Else /* relation_tag represents S*/

▷ $v \leftarrow$ apply the aggregate function on values of List_V ;

End If

▷ Emit (K,v);

}

Algorithm 9 Map function: /* To generate relevant randomized tagged records for $Hist_i^{x,y}(\overline{R})$ and $AGGR_{f(u),i}^{x,z}(\overline{S})$ using $HistIndex$ communication templates.*/

```
void map_init(){
  inner_tag ← 1 ;                /* a tag to identify relation R records */
  outer_tag ← 2 ;                /* a tag to identify relation S records */
  hash_index ← 0;                /* a flag to identify hash based records */
  partition_index ← 1;           /* a flag to identify records to partition */
  replicate_index ← 2 ;          /* a flag to identify records to replicate */
  Read HistIndex(R ⋈ S): histogram index from DFS;
  Create a HashTable using join_key value, frequency's index and Nb_buckets of HistIndex(R ⋈ S);
}
map(K: null, V : a record from a split of either relation Hist_i^{x,y}(R̄) or AGGR_{f(u),i}^{x,z}(S̄)) {
  ▷ relation_tag ← get relation tag from current relation split;
  ▷ join_key ← extract the join value from record V of current input relation;
  ▷ groupby_value ← extract the group-by attribute from record V of current input relation;
  ▷ aggregated_value ← extract the partial computed aggregate value from record V of current
  input relation;
  If (join_key ∈ HashTable) Then /* To redistribute only relevant records */
    ▷ frequency_index ← HashTable(join_key).frequency_index;
    ▷ Nb_buckets1 ← HashTable(join_key).Nb_buckets1;
    ▷ Nb_buckets2 ← HashTable(join_key).Nb_buckets2;
    ▷ random_integer ← Generate_Random_Integer(join_key);
    If (frequency_index = hash_index) Then
        ▷ Emit ((join_key,-1,relation_tag), (groupby_value, aggregated_value)); /* for records, with
                  low frequencies, to be hashed */
    ElseIf (((frequency_index = partition_index) and (relation_tag = inner_tag))
                  or ((frequency_index = replicate_index) and (relation_tag=outer_tag)))
        ▷ random_dest ← (random_integer+SRAND(Nb_buckets1)) % Nb_buckets1;
          /* A random integer between 0 and Nb_buckets1 */
        ▷ flag_index ← partition_index ;
        ▷ Emit ((join_key,random_dest,(flag_index,relation_tag)), (groupby_value, aggregated_value));
    Else
        For (int i=0; i<Nb_buckets1; i++) Do
          ▷ random_dest ← (random_integer+i) % Nb_buckets1;
          ▷ flag_index ← replication_index ;
          ▷ bucket_dest ← i % Nb_buckets2;    /* A random integer between 0 and Nb_buckets2 */
          ▷ Emit ((join_key,random_dest,(flag_index,relation_tag,bucket_dest)),
                                              (groupby_value, aggregated_value));
        End For;
    End If;
  End If;
}
```

Algorithm 10 Partitioning function /* Returns for each composite input key K =(join_key,random_integer,DataTags) emitted in Map phase, an integer corresponding to destination reducer for key K. */

```
int partition(K: input key ){
   join_key ← K.join_key;              /* extracts join key part from input key K */
   relation_tag ← K.relation_tag;     /* extracts relation tag part from input key K */
   reducer_dest ← K.random_dest; /* extracts reducer destination number from input key K */
   If (reducer_dest ≠ -1) Then
     Return (reducer_dest % NB_reducers);
   Else
     Return (HashCode(join_key) % NB_reducers);
   End If ;
}
```

Algorithm 11 Reduce function /* To generate groupby and join result. */

```
void reduce_init(){
  last_key ← "" ;                          /* to store the last processed key */
  inner_relation_tag ← 1 ;                 /* a tag to identify Inner relation records */
  outer_relation_tag ← 2 ;                 /* a tag to identify Outer relation records */
  Array_buffer ← NULL ;                    /* an array list used to buffer records from one relation */
}
reduce(Key K,List List_V ) {    /* List List_V: the list of records from either Hist_i^{x,y}(R)
                        or AGGR^{x,z}_{f(u),i}(S)) */
  ▷ join_key ← K.join_key;                    /* extracts the join key part from input key K */
  ▷ groupby_attribute ← K.groupby_attribute;
                                    /* extracts the group by attribute part from input key K */
  ▷ relation_tag ← K.relation_tag;          /* extracts relation tag part from input key K */
  ▷ flag_index ← K.flag_index;              /* extracts flag index part from input key K */
  If ((join_key = last_key) and (relation_tag ≠ flag_index)) Then
     For each record (x ∈ List_V) Do
       For each record (y ∈ Array_buffer) Do
          If (relation_tag = outer_relation_tag) Then
            ▷ Emit (NULL, x ⊕ y);
          Else
            ▷ Emit (NULL, y ⊕ x);
          End If ;
       End For ;
     End For ;
  Else
    ▷ Array_buffer.Clear();
    For each record (x ∈ List_V) Do
        ▷ Array_buffer.Add(x);
    End For ;
    ▷ last_key ← K.join_key;
  End if
}
```