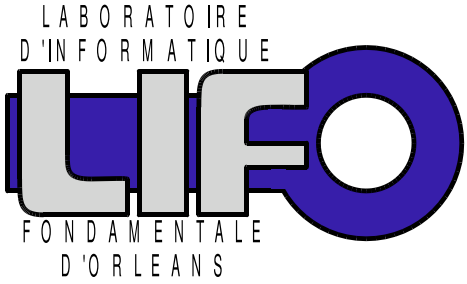

4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE
http://www.univ-orleans.fr/lifo

Rapport de Recherche

A Scalable and Skew-insensitive Algorithm for Join Operations using Map/Reduce Model

Mostafa BAMHA, Frédéric Loulergue
LIFO, Université d'Orléans

Rapport n^o **RR-2014-01**

A Scalable and Skew-insensitive Algorithm for Join Operations using Map/Reduce Model

Mostafa Bamha and Frédéric Loulergue

Université Orléans, INSA Centre Val de Loire, LIFO EA 4022, France
Mostafa.Bamha@univ-orleans.fr
Frederic.Loulergue@univ-orleans.fr

Abstract

For over a decade, Map/Reduce has become a prominent programming model to handle vast amounts of raw data in large scale systems. This model ensures scalability, reliability and availability aspects with reasonable query processing time. However these large scale systems still face some challenges : data skew, task imbalance, high disk i/o and redistribution costs can have disastrous effects on performance.

In this paper, we introduce *MRFA-Join* algorithm: a new Frequency Adaptive algorithm based on Map/Reduce Programming model and distributed histograms for join processing on large-scale datasets. A cost analysis of this algorithm shows that our approach is insensitive to data skew and ensures perfect balancing properties during all stages of join computation. Performances have been experimented on Grid'5000 infrastructure.

Keywords: Join operations, Data skew, Map/Reduce model, Hadoop framework.

1 Introduction

Join operation is one of the most widely used operations in relational database systems, but it is also a heavily time consuming operation. For this reason it was a prime target for parallelization. The *join* of two relations R and S on attribute A of R and attribute B of S (A and B of the same domain) is the relation, written $R \bowtie S$, obtained by concatenating the pairs of tuples from R and S for which $R.A = S.B$.

Parallel join usually proceeds in two phases: a redistribution phase (generally based on join attribute hashing) and then a sequential join of local fragments. Many parallel join algorithms have been proposed. The principal ones are: *Sort-merge join*, *Simple-hash join*, *Grace-hash join* and *Hybrid-hash join* [15]. All of them (called hashing algorithms) are based on hashing functions which redistribute relations such that all the tuples having the same attribute value are forwarded to the same node. Local joins are then computed and their union is the output relation. Research has shown that join is parallelizable with near-linear speed-up on distributed architectures but only under ideal balancing conditions : data skew may have disastrous effects on the performance [13, 16]. To this end, several parallel algorithms were presented to handle data skew while treating join queries on parallel database systems [1–3, 8, 13, 16].

Today with the rapid development of network technologies, Internet search engines and Data Mining applications, the need to manage and query a huge amount of datasets every day becomes essential. Parallel processing of such queries on hundreds or thousands of nodes is obligatory to obtain a reasonable processing time [7]. However, building parallel programs on parallel and distributed systems is complicated because programmers must treat several issues such as load balancing, fault tolerance, etc.

Search engine companies have developed Distributed File Systems (DFS) and parallel programming infrastructures that treat these parallel processing related issues without the explicit

participation of the programmers [11]. Hadoop [10], Google's MapReduce model [11], Google file system [9], BigTable [6] are examples of such systems. These systems are built from thousands of commodity machines and assure scalability, reliability and availability aspects [12]. To reduce disk i/o, each file in such storage systems is divided into chunks or blocks of data and each block is replicated on several nodes for fault tolerance. Parallel programs are easily written on such systems following the Map/Reduce paradigm where a program is composed of a workflow of user defined *map* and *reduce* functions [7, 12]. The *map* function operates on a (key, value) couple and produces intermediate key with an associated list of values. The *reduce* function merges all the intermediate couples having the same key.

In this paper we are interested in the evaluation of join operations on large scale systems using Map/Reduce model. This programming model is designed to simplify the development of large-scale, distributed, fault-tolerant data processing applications.

In [17], three well known algorithms for join evaluation were implemented using an extended Map/Reduce model. These algorithms are *Sort-Merge-Join*, *Hash-Join* and *Block Nested-Loop Join*. Combining this model with DFS facilitates the task of programmers because they don't need to take care of fault tolerance and load balancing issues. However, load balancing in the case of join operations is not straightforward in the presence of data-skew. In [4] Blanas & all. have presented an improved versions of MapReduce sort-merge joins and semi-join algorithms for Log processing, to fix the problem of buffering all records from both inner and outer relations in "Standard repartition join" provided in Hadoop Contributed join package (org.apach.hadoop.contrib). For the same reasons as in PDBMS[1], even in the presence of integrated functionality for load balancing and fault tolerance in MapReduce, these algorithms still suffer from the effect of data skew since all tuples having the same join values in Map phase are sent to the same reducer which limits the scalability of the presented algorithms [12].

The aim of join operations is to combine information from two or more data sources, Unfortunately, MapReduce framework is somewhat inefficient to perform such operations since data from one source must be maintained in memory for comparison to other source of data. Consequently, adapting well-known join algorithms to MapReduce is not as straightforward as one might hope, and MapReduce programmers often use simple but inefficient algorithms to perform join operations especially in the presence of skewed data [4, 12, 14].

To avoid the effect of data skew in join operations using Map/Reduce model, we introduce *MRFA-Join* (Map/Reduce Frequency Adaptive Join) algorithm based on distributed histograms and randomized key's redistribution approach. This algorithm, inspired from our previous research on Join and Semi-join operations in PDBMS, is well adapted to manage huge amount of data on large scale systems even for highly skewed data.

2 The Map-Reduce Programming Model

Google's Map/Reduce programming model presented in [7] is based on two functions: *Map* and *Reduce*. Dean and Ghemawat stated that they have inspired their Map/Reduce model from Lisp and other functional languages [7]. The programmer is only required to implement two functions *Map* and *Reduce* having the following signatures:

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

[1] PDBMS: Parallel Database Management Systems.

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 Map/Reduce 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 $list(v_3)$.

Map/Reduce 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.

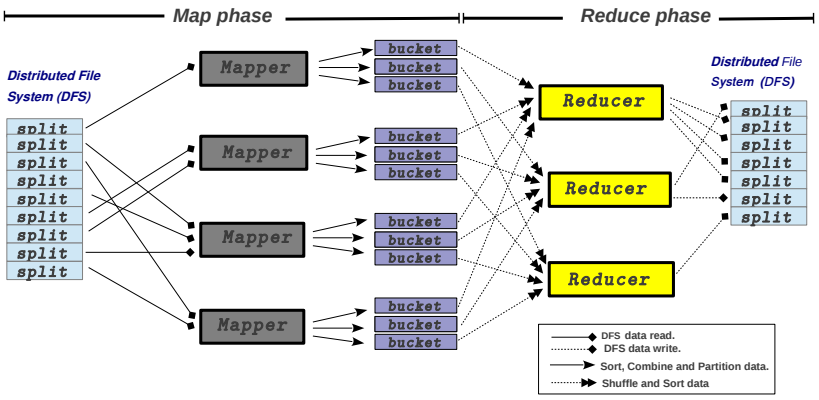

Figure 1: Map-reduce framework.

In this paper, we used an open source version of Map/Reduce called Hadoop developed by "The Apache Software Foundation". Hadoop framework includes a distributed file system called HDFS[2] designed to store very large files with streaming data access patterns.

For efficiency reasons, in Hadoop Map/Reduce 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:

combine: $(k_2, list(v_2)) \longrightarrow (k_2, list(v_3))$.

To cover a large range of applications need 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 (Hadoop's default "partition function" is based on "hashing" the whole input key k_2). The signature of the partition function is :

[2]HDFS: Hadoop Distributed File System.

partition: $(Key \quad k_2) \longrightarrow Integer$. `/* Integer is between 0 and the number of reducers #numReduceTasks */`

3 MRFA-Join: A Map-Reduce Skew Insensitive Join Algorithm

As stated in the introduction section, Map/Reduce hash based join algorithms presented in [4, 17] may be inefficient in the presence of highly skewed data [14] due to the fact that in Map function in these algorithms, all the key-value pairs (k_1, v_1) representing the same entry for the join attribute are sent to the same reducer (In Map phase, emitted key-value pairs (k_2, v_2), the key k_2 is generated by only using join attribute values in the manner that all records with the same join attribute value will be forwarded to the same reducer).

To avoid the effect of repeated keys, Map user-defined function should generate distinct output keys k_2 even for records having the same join attribute value. This is made possible by using a user defined partitioning function in Hadoop : the reducer destination for a key k_2 can be computed from different parts of key k_2 and not by a simple hashing of all input key k_2. To this end, we introduce, in the next section, a join algorithm called MRFA-Join (Map/Reduce Frequency Adaptive Join) based on distributed histograms and a random redistribution of repeated join attribute values combined with an efficient technique of redistribution where only relevant data is redistributed across the network during the shuffle phase of reduce step. A cost analysis for MRFA-Join is also presented to give for each computation step, an upper bound of execution time in order to prove the strength of our approach.

In this section, we describe the implementation of MRFA-Join 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 Map/Reduce framework.

To compute the join, $R \bowtie S$, of two relations (or datasets) R and S, we assume that input relations R and S are divided into blocks (splits) of data. These splits are stored in Hadoop Distributed File System (HDFS). 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 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,
- $\overline{T_i}$: the split(s) of relation $\overline{T}$ affected to mapper i,
- $||T_i||$: number of tuples in split T_i,
- $Hist^{map}(T_i^{map})$: Mapper's local histogram of T_i^{map}, i.e. the list of pairs (v, n_v) where v is a join attribute value and n_v its corresponding frequency in relation T_i^{map} on mapper i,
- $Hist_i^{red}(T)$: the fragment of global histogram of relation T on reducer i,
- $Hist_i^{red}(T)(v)$ is the global frequency n_v of value v in relation T,
- $HistIndex(R \bowtie S)$: join attribute values that appear in both R and S and their corresponding three parameters: *Frequency_index*, *Nb_buckets* and *random_int* used in communications 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 of 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 MRFA-Join algorithm while giving a cost analysis for each computation phase. Join computation in MRFA-Join proceeds in two map-reduce jobs:

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

b. the second one, is used to generate join output result by using communications templates carried out in the previous step.

In the following, we will describe MRFA-Join steps while giving an upper bound on the execution time of each map-reduce step. The O(...) notation only hides small constant factors: they only depend program's implementation but neither on data nor on the machine parameters. Data redistribution in MRFA-Join algorithm is the basis for efficient and scalable join processing while avoiding the effect of data skew in all the stages of join computation. MRFA-Join algorithm proceeds in 4 steps:

a.1 Map phase to generate a tagged "local histogram" for input relations:
In this step, each mapper i reads its assigned data splits (blocks) of relation 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 join key value 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 reducers destination in the shuffle phase of the the following reduce step. The cost of this step is at most : $Time(a.1.2) =$

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

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

We recall that, in this step, only local histograms $Hist^{map}(R_i^{map})$ and $Hist^{map}(S_i^{map})$ are sorted and transmitted across the network and the size 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, $Hist^{map}(T)$ contains only distinct entries of the form (v, n_v) where v is a join attribute value and n_v the 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 $Hist_i^{red}(R)$

(resp. $Hist_i^{red}(S)$) obtained through hashing of distinct values of $Hist^{map}(R_j^{map})$ (resp. $Hist^{map}(S_j^{map})$) of each mapper j. Received $Hist_i^{red}(R)$ and $Hist_i^{red}(S)$ 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 the 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 of 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 $K \in HistIndex_i(R \bowtie S)$,
  - $Frequency\_index(K) = 0$, means that the frequency of tuples of relations R and S, associated to value K, are less than the threshold frequency: $Hist_i^{red}(R)(K) < f_0$ and $Hist_i^{red}(S)(K) < f_0$,
  - $Frequency\_index(K) = 1$, means that the frequency of tuples of relation R having the value K are greater than the threshold frequency, and the frequency in relation R for this attribute value is greater than the corresponding frequency in relation S: $Hist_i^{red}(R)(K) \geq f_0$ and $Hist_i^{red}(R)(K) \geq Hist_i^{red}(S)(K)$.
  - $Frequency\_index(K) = 2$, means that the frequency of tuples of relation S associated to value K are greater than the threshold frequency, and the frequency in relation S for this attribute value is greater than the corresponding frequency in relation R: $Hist_i^{red}(S)(K) \geq f_0$ and $Hist_i^{red}(S)(K) > Hist_i^{red}(R)(K)$,
- 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.
Using this information, each reducer i, has local knowledge of how relevant records of input relations will be redistributed in the next following map phase.
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 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 hash table and to redistribute relevant data using randomized communication templates:
In this step, each mapper i reads join result global histogram index, $HistIndex$, to create a local Hash table in time: $Time(b.1.1) = O(\max_{i=1}^{NB\_mappers} t_h^i * ||HistIndex(R \bowtie S)||)$.
Once the local hash table 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 hash table) or routed to a designated random reducer destination using communication templates computed in step a.2 (Map phase details are described in Algorithm 6).
Figure 2 gives an example of communication templates used to partition data for

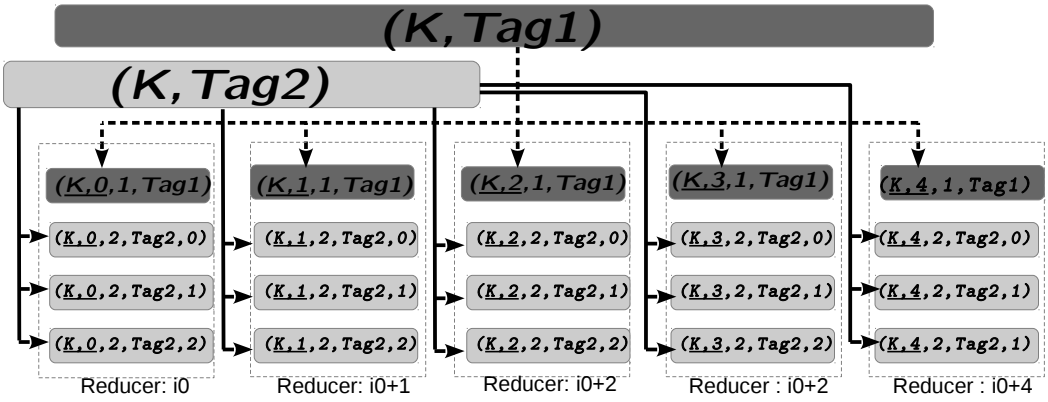

Figure 2: Generated buckets associated to a join key K corresponding to a high frequency where records from relation associated to Tag_1 (i.e relation having the highest frequency) are partitioned into five buckets and those of relation associated to Tag_2 are partitionned into three buckets.

(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. $Nb\_buckets1(K) = 5$) where as 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.
The global cost of this step is at most:

$$Time_{step_{a.2}} = O\big(\max_{i=1}^{NB\_reducers}(||Hist_i^{red}(R)|| + ||Hist_i^{red}(S)||)\big).$$

Note that, $HistIndex(R \bowtie S) \equiv \cup_i(Hist_i^{red}(R) \cap Hist_i^{red}(S))$ and $||HistIndex(R \bowtie S)||$ is very small compared to $||Hist^{red}(R)||$ and $||Hist^{red}(S)||$.
The cost of this step is :

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

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 hash table search for each input record, $||\overline{R}_i^{map}|| * \log||\overline{R}_i^{map}|| + ||\overline{S}_i^{map}|| * \log||\overline{S}_i^{map}||$ is time to sort relevant data on mapper i, where as the term $c_{comm} * (|\overline{R}_i^{map}| + |\overline{S}_i^{map}|))$ 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:

$$Time_{step_{b.1}} = Time(b.1.1) + Time(b.1.2).$$

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.

b.2 Reduce phase to compute join result:
At the end of step b.1, each reducer i receives a fragment $\overline{R}_i^{red}$ (resp. $\overline{S}_i^{red}$) obtained through randomized hashing of $\overline{R}_j^{map}$ (resp. $\overline{S}_j^{map}$) of each mapper j and performs a local join of received data. This reduce phase is described in detail in Algorithm 8. The cost of this step is:

$$Time_{step_{b.2}} = O(\max_{i=1}^{NB\_reducers}(||\overline{R}_i^{red}|| + ||\overline{S}_i^{red}|| + c_{r/w} * |\overline{R}_i^{red} \bowtie \overline{S}_i^{red}|).$$

The global cost of MRFA-Join is therefore the sum of the above four steps :

$$Time_{MRFA-Join} = Time_{step_{a.1}} + Time_{step_{a.2}} + Time_{step_{b.1}} + Time_{step_{b.2}}$$

Using hashing technique, the join computation of $R \bowtie S$ requires at least the following lower bound : $bound_{inf} =$

$$\Omega\Big(\max_{i=1}^{NB\_mappers}\big((c_{r/w} + c_{comm}) * (|R_i^{map}| + |S_i^{map}|) + ||R_i^{map}|| * \log||R_i^{map}|| + ||S_i^{map}|| * \log||S_i^{map}||\big) + \max_{i=1}^{NB\_reducers}\big(||R_i^{red}|| + ||S_i^{red}|| + c_{r/w} * |R_i^{red} \bowtie S_i^{red}|\big)\Big),$$

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 on 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 and $c_{r/w} * |R_i^{red} \bowtie S_i^{red}|$ represents the cost to store reducer's i join result on the DFS.

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

$$\leq \max\left(\max_{i=1}^{NB\_mappers}(||R_i^{map}|| * \log||R_i^{map}||, |S_i^{map}|| * \log||S_i^{map}||), \max_{i=1}^{NB\_reducers}||R_i^{red} \bowtie S_i^{red}||\right), \quad (1)$$

this is due to the fact that, all other terms in $Time_{MRFA-Join}$ 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 [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. There is no way for a clever hash function to avoid load imbalance that results from these repeated values [8]. 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 randomized keys.

4 Experiments

To evaluate the performance of `MRFA-Join` algorithm presented in this paper, we compared our algorithm to the best known solutions called respectively `Improved_Repartition_Join` and `Standard_Repartition_Join`. `Improved_Repartition_Join` was introduced by Blanas et al. in [4], where as `Standard_Repartition_Join` is the join algorithm provided in Hadoop framework's contributions. 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 1. 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.

Table 1: 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

To study the effect of data skew on performance, join attribute values in the generated data have been chosen to follow a Zipf distribution [18] as it is the case in most database tests: Zipf factor was varied 0 (for a uniform data distribution) to 1.0 (for a highly skewed data). Input relations size was fixed to 200M records for the right relation (approximately 20GB of data) and 10M of records for the left relation (approximately 1GB of data) and the join result varying from approximately 40M to 1700M records (corresponding respectively to about 8GB and 340GB of output data).

We noticed in all the tests and also those presented in figure 3, that our `MRFA-Join` algorithm outperforms both `Improved_Repartition_Join` and `Standard_Repartition_Join` algorithms even for low or moderated skew. We recall that our algorithm requires the scan of input data twice : the fist one for histogram processing and the second scan is performed for join processing. The cost analysis and tests performed showed that the overhead related to histogram processing is compensated by the gain in join processing since only relevant data (that appears in the join result) is emitted by mappers in the map phase which reduce considerably the amount of data transmitted over the network in shuffle phase (see figure 4). Moreover, for skew factors varying from 0.6 to 1.0, both `Improved_Repartition_Join` and `Standard_Repartition_Join` jobs fails

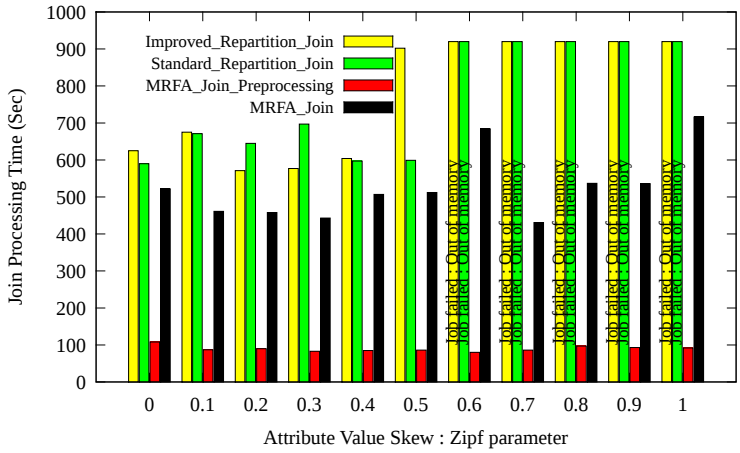

Figure 3: Data skew effect on Hadoop join processing time

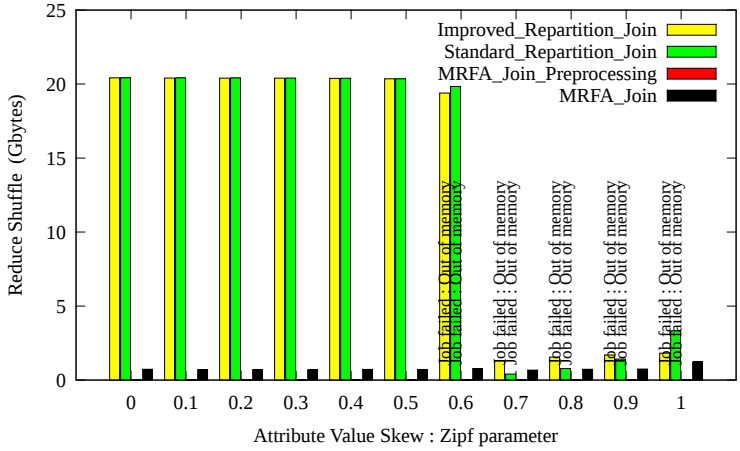

Figure 4: Data skew effect on the amount of data moved across the network during Shuffle phase

due to memory lack this is due to the fact that, in reduce phase, all the records emitted by the mappers having the same join key are send and processed by the same reducer which makes both `Improved_Repartition_Join` and `Standard_Repartition_Join` algorithms very sensitive to data skew and limits their scalability. This can not occur in MRFA-Join owing to the fact that attribute values associated to high frequencies are forwarded to distinct reducers using randomized join attribute keys and not by a simple hashing of record's join key. We expect a higher gain related to histograms pre-processing in complex queries computation due to the fact that histograms can be used to reduce drastically the costs of communication and disk input/output of intermediate data by generating only relevant data for each sub-query.

5 Conclusion and future work

In this paper, we have introduced the first skew-insensitive join algorithm, called MRFA-Join, using Map/Reduce model, based on distributed histograms and randomized keys redistribution for highly skewed data. The detailed information provided by these histograms, allows us to reduce communications costs to only relevant data while guaranteeing perfect balancing processing due to the fact that, all the generated join tasks and buffered data do not never exceed a user defined size using threshold frequencies. This makes the algorithm scalable and outperforming existing map-reduce join algorithms which fail to handle skewed data whenever a join task can not fit in the available node's memory. We mention that MRFA-Join can also benefit from Map/Reduce underlying load balancing framework in heterogeneous or a multi-user environment since MRFA-Join is implemented without any change in Map/Reduce framework.

Our experience with the join operations shows that the overhead related to distributed histograms processing remains very small compared to the gain in performance and communication costs since only relevant data is processed or redistributed across the network.
Future work will consist in studying how distributed histograms could be used to compute more complex or pipelined join queries to reduce communication and disk input/output costs for intermediate data for each sub-query by generating only relevant data.

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`). This work is also partly supported under the INEX project funded by the *Conseil Général du Loiret*.

References

[1] M. Bamha. An optimal and skew-insensitive join and multi-join algorithm for distributed architectures. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA'2005). 22-26 August, Copenhagen, Danemark*, volume 3588 of *LNCS*, pages 616–625. Springer, 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*, pages 644–653, London, United Kingdom, 2000. Springer-Verlag.

[3] 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.

[4] 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 the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 975–986, New York, NY, USA, 2010. ACM.

[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] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system

for structured data. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 205–218, Berkeley, CA, USA, 2006. USENIX Association.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI ' 04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA*, 2004.

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

[9] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM Press.

[10] Apache hadoop. http://hadoop.apache.org/core/.

[11] Ralf Lämmel. Google's mapreduce programming model — revisited. *Science of Computer Programming*, 68(3):208–237, 2007.

[12] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. Parallel data processing with mapreduce: A survey. *ACM SIGMOD Record*, 40(4):11–20, December 2011.

[13] 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.

[14] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. Dewitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *In SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 165–178. ACM, 2009.

[15] D. Schneider and D. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In J. Clifford, B. Lindsay, and D. Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data, Portland, Oregon*, pages 110–121, New York, NY 10036, USA, 1989. ACM Press.

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

[17] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040, New York, NY, USA, 2007. ACM.

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

6 Appendix: Implementation of MRFA-Join functions

Algorithm 1 MRFA-join algorithm workflow.

a.1► Map phase: /* To generate a tagged "Local histogram" for input relations */
▷ Each mapper i reads its assigned data splits (blocks) of relation R_i^{map} and S_i^{map} from the DFS
▷ Extract the `join_key` value from input relation's record.
▷ Get a `tag` to identify source input relation.
▷ Emit a couple ((join_key,tag),1) /* `a tagged join_key` with a frequency 1 */
► Combine phase: To compute local frequencies for each join_key value in relations R_i^{map} and S_i^{map}
▷ 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► Reduce phase: /* To combine Shuffle's output records and to create Global join histogram index */
▷ Compute the global frequencies for only join_key values present in both relations R and S.
▷ Emit, for each join_key, a couple (join_key,(frequency_index,Nb_buckets1, Nb_buckets2)).
/* frequency_index $\in \{0, 1, 2\}$ used to get detailed information about data distribution in R and S */
b.1► Map phase:
▷ Each mapper, i, reads join result Global histogram index from DFS, and creates a local Hashtable.
▷ Each mapper, i, reads its assigned data splits (blocks) of relation R_i^{map} and S_i^{map} from DFS, and generates randomized communication templates for records in R_i^{map} and S_i^{map} according to join_key value and its corresponding frequency_index in HashTable. In communication templates, only relevant records from R_i^{map} and S_i^{map} are emitted using a hash or a randomized partition/replicate schema. Emit relevant randomised tagged records from relations R_i^{map} and S_i^{map}.
► Partition phase:
▷ For each emitted tagged join_key, compute reducer destination according to the value of join_key, and reducer random destination generated in Map phase;
b.2► Reduce phase: to combine Shuffle's output records and to generate join result

Algorithm 2 Map function /* To generate local histograms 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;
▷ Emit ((join_key,relation_tag), 1);
}

Algorithm 3 Combine function: /* To compute local histogram's frequencies for join_key */

combine(Key K,List List_V) { /* List_V is the list of values "1" corresponding to the unique frequencies in relation R_i or S_i emitted by Mappers */
▷ frequency ← sum of frequencies in List_V ;
▷ Emit (K,frequency);
}

Algorithm 4 Partitioning function /* Returns for, each composite key K=(join_key,relation_tag) emitted in Map phase, an integer corresponding to destination reducer for the input key K. */

```
int partition(K: input key ){
    ▷ join_key ← K.join_key;            /* extracts join key part from input key K */
    ▷ Return (HashCode(join_key) % NB_reducers);
}
```

Algorithm 5 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 6 Map function: /* To generate relevant randomized tagged records for input relations 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;
  Build HashTable: a hash table using join_key values and frequency's index present in 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
    ▷ frequency_index ← HashTable(join_key).frequency_index; /* gets join_key's frequency index from HashTable */
    ▷ Nb_buckets1 ← HashTable(join_key).Nb_buckets1; /* extracts Nb_buckets1 for join_key from HashTable */
    ▷ Nb_buckets2 ← HashTable(join_key).Nb_buckets2; /* extracts Nb_buckets2 for join_key from HashTable */
    ▷ random_integer ← Generate_Random_Integer(join_key); /* generates a random integer for join_key */
    If (frequency_index = hash_index) Then
        Emit ((join_key,-1,relation_tag, V); /* reducer_dest=-1 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;
        /* To generate a random integer between 0 and Nb_buckets1 */
        Emit ((join_key,random_dest,(partition_index,relation_tag)), V);
    Else
        For (int i=0; i<Nb_buckets1; i++) Do
          random_dest ← (random_integer+i) % Nb_buckets1;
          bucket_dest ← i % Nb_buckets2;
          Emit ((join_key,random_dest,(replicate_index,relation_tag,bucket_dest)), V);
        End For ;
    End If ;
  End If ;
}
```

Algorithm 7 Partitioning function /* Returns for each composite input key K =(join_key,random_integer,DataTag) 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 8 Reduce function: /* to generate 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 relation R or S */
  ▷ join_key ← K.join_key;                 /* extracts the join key part from input key K */
  ▷ relation_index ← K.relation_index;     /* extracts relation index part from input key K */
  ▷ relation_tag ← K.relation_tag;         /* extracts relation tag part from input key K */
  If ((join_key = last_key) and (relation_index ≠ inner_relation_tag)) 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
}
```