Procedia Computer Science
Volume 51, 2015, Pages 70–79
ICCS 2015 International Conference On Computational Science

Towards Scalability and Data Skew Handling in GroupBy-Joins using MapReduce Model

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

[1] Lebanese International University, Beirut, Lebanon
mohamad.hajjhassan01@liu.edu.lb
[2] Université Orléans, INSA Centre Val de Loire, France
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 frameworks such as Spark, Pig and 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 GrouBy-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 [2, 10, 14]. 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.

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2015

doi:10.1016/j.procs.2015.05.200

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.

The difference between queries $\mathcal{Q}_1$ 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 [4, 15]. But the response time of these queries may be significantly reduced if the group-by operation and aggregate function are performed before the join [4]. 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 [4, 7, 15, 16]. 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 [8, 9, 13]. 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 [11, 12].

We have recently proposed in [10], *MRFA-Join* algorithm, a MapReduce based algorithm for evaluating join operations on Distributed File Systems (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 [2, 17], 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 processor 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 [5] 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 [5] is based on two functions: **map** and **reduce**. 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 used an open source version of MapReduce called Hadoop developed by "The Apache Software Foundation". Hadoop framework includes a distributed file system called Hadoop Distributed File System (HDFS) 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. It is like a local reduce applied (at map worker) before storing or sending intermediate results to the reducers. Its signature is: **combine:** $(k_2, list(v_2)) \longrightarrow (k_2, list(v_3))$.

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: **partition:** $k_2 \longrightarrow Integer$, 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.

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 HDFS 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 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:

- $\overline{T}$: the restriction (a fragment) of relation T which contains tuples that appear in the final GroupBy-join result,
- T_i^{map} : the split(s) of relation T affected to mapper i,

- $\overline{T_i}$: the split(s) of relation $\overline{T}$ affected to mapper 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,
- $AGGR^w_{f(u)}(T_i)$: 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. $AGGR^w_{f(u)}(T_i)$ 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),
- $AGGR^w_{f(u),i}(T)$: is the fragment of $AGGR^w_{f(u)}(T)$ 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.

MRFAG-Join proceeds in three MapReduce jobs (refer to [1] for a cost analysis of each phase):

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.

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 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 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. 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. 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

Algorithm 1 MRFAG-join algorithm workflow for query $\mathcal{Q}_1$

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 $\in \{0, 1, 2\}$: 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 $AGGR^{x,z}_{f(u)}(\overline{S}_i^{map})$ */
▷ 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 $\in$ HashTable) **Then**
  ▷ Extract the groupby attribute y of R (resp. z of S in addition to 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 $AGGR^{x,z}_{f(u),i}(\overline{S})$ */
▷ 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 $AGGR^{x,z}_{f(u),i}(\overline{S})$ */
▷ 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 $AGGR^{x,z}_{f(u),i}(\overline{S})$ from DFS,
▷ Generates randomized communication templates for records in $Hist_i^{x,y}(\overline{R})$ and $AGGR^{x,z}_{f(u),i}(\overline{S})$ 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 $AGGR^{x,z}_{f(u),i}(\overline{S})$.
▶ 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, $AGGR^{x,y,z}_{f(u),i}(R \bowtie S)$, on each reducer i.

$HistIndex_i(R \bowtie S)$ 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 $K \in HistIndex_i(R \bowtie S)$,

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

- 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.

For a highly skewed join attribute value K, appropriate map keys are generated so that all records in each bucket associated to value K in one relation are forwarded to the same reducer holding all the corresponding buckets of other relation. 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 [10].

Using $HistIndex$ information, each reducer i, has local knowledge of how relevant records of input relations will be redistributed in the next 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 *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. Once local hashtable is created on each mapper, input relations are then read from DFS, and each input record is either discarded (if record's join key value 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 details are described in Algorithm 5 of appendix in [1]).

b.2: Reduce phase to compute relevant partial aggregated data, $Hist_i^{x,y}(\overline{R})$ and $AGGR^{x,z}_{f(u),i}(\overline{S})$:

At the end of step b.1, each reducer i receives a fragment $Hist_i^{x,y}(\overline{R})$ (resp. $AGGR^{x,z}_{f(u),i}(\overline{S})$) obtained through a hashing of $Hist^{x,y}(\overline{R}_j^{map})$ (resp. $AGGR^{x,z}_{f(u)}(\overline{S}_j^{map})$) on each mapper j and

Relation : R

x	y	v
1	1	1
1	1	1
1	3	1
1	4	1
2	2	1
2	2	1
2	5	1
3	7	1
3	10	1
4	1	1
4	2	1
5	4	1
6	1	1
6	2	1
6	3	1
7	1	1
7	2	1
7	3	1
8	9	1
9	1	1
9	1	1
9	1	1
9	1	1
9	1	1
9	2	1
9	2	1
9	3	1

Relation : S

x	z	u
1	4	1
1	4	2
1	3	3
2	1	4
2	1	5
2	1	6
2	3	7
2	5	8
6	1	9
7	2	10
9	5	11
9	6	12
9	7	13
10	1	1
10	1	2
10	1	3
10	1	4
10	2	5
10	3	6

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

x	y	freq
1	1	2
1	3	1
1	4	1
2	2	2
2	5	1
~~3~~	~~7~~	~~1~~
~~3~~	~~10~~	~~1~~
~~4~~	~~1~~	~~1~~
~~4~~	~~2~~	~~1~~
~~5~~	~~4~~	~~1~~
6	1	1
6	2	1
6	3	1
7	1	1
7	2	1
7	3	1
~~8~~	~~9~~	~~1~~
9	1	5
9	2	2
9	3	1

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

x	z	$f^{x,z}(u)$
1	4	3
1	3	3
2	1	15
2	3	7
2	5	8
6	1	9
7	2	10
9	5	11
9	6	12
9	7	13
~~10~~	~~1~~	~~10~~
~~10~~	~~2~~	~~5~~
~~10~~	~~3~~	~~6~~

GroupBy − join result : $AGGR^{x,y,z}_{f(u)}(R \bowtie S)$

x	y	z	$f^{x,y,z}(u)$
1	1	4	6
1	1	3	6
1	3	4	3
1	3	3	3
1	4	4	3
1	4	3	3
2	2	3	14
2	2	1	30
2	2	5	16
2	5	1	15
2	5	3	7
2	5	5	8
6	1	1	9
6	2	1	9
6	3	1	9
7	1	2	10
7	2	2	10
7	3	2	10
9	1	5	55
9	1	6	60
9	1	7	65
9	2	5	22
9	2	6	24
9	2	7	26
9	3	5	11
9	3	6	12
9	7	3	13

Query Q1: **Select** R.x, R.y, S.z, SUM(S.u)
From R, S
Where R.x = S.x
Group By R.x, R.y, S.z

In this example function 'f' is the function 'SUM'.

$f^{w}(u)$: is the aggregate function 'f' applied to attribute 'u' for all tuples having the same value for attribute 'w' (w may be formed from more than one attribute).

Figure 1: Final GroupBy-Join computation using $Hist^{x,y}(\overline{R})$ and $AGGR^{x,z}_{f(u)}(\overline{S})$.

then a local merge of received data. This partial aggregated data is then written to DFS. (refer to Algorithm 8 of appendix in [1] for a complete description of this reduce phase).

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. 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 (refer to Algorithm 9 of Appendix in [1] for a detailed description of Map phase).

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. $AGGR^{x,z}_{f(u),i}(\overline{S})$) obtained through randomized hashing of $\overline{R}_j^{map}$ (resp. $\overline{S}_j^{map}$) on each mapper j and performs a local GroupBy-join of received data. Figure 1 shows how the frequency in $Hist^{x,y}(\overline{R})$ and the value of aggregate function in $AGGR^{x,z}_{f(u)}(\overline{S})$ are used to generate final `GroupBy-Join` result.

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 [3],
- *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 [6]. 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 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 [18] 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 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 2, that our `MRFAG-Join` (including histograms and aggregate data preprocessing) algorithm outperforms `PigLatin_GroupBy_Join` execution even for low or moderated skew (∼ 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 2, 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 (see [1]) 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 3). 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 join key values.

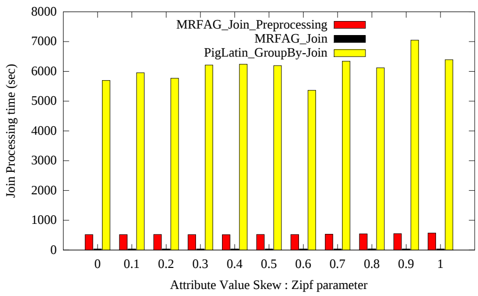

Figure 2: Data skew effect on Hadoop GroupBy-join processing time

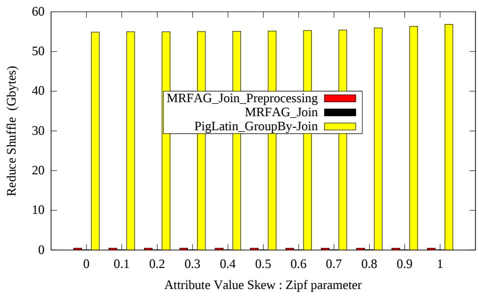

Figure 3: Data skew effect on data volume moved across the network during shuffle phase

5 Conclusion and future work

In this paper, we have presented MRFAG-Join a 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 related 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] M. Bamha and M. Al Hajj Hassan. Scalability and optimisation of groupby-joins in mapreduce. Technical report, LIFO, Université d'Orléans, France., March 2015.

[2] 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.

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

[4] 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.

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

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

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] D. Taniar, Y. Jiang, K.H. Liu, and C.H.C. Leung. Aggregate-join query processing in parallel database systems,. In *the 4th International Conference/Exhibition on High Performance Computing in Asia-Pacific Region*, volume 2, pages 824–829. IEEE Computer Society, 2000.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] H. Yang, A. Dasdan, R. Hsiao, and D. 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.

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