EVALUATION OF JOIN AND JOIN BASED BLOOM FILTER ALGORITHMS IN MAPREDUCE ENVIRONMENT

RAJAONARIVELO Maminiaina Andry Tahiana [1], RAKOTOMIRAHO Soloniaina[2], RANDRIAMAROSON Rivo Mahandrisoa[3]

[1] PhD student, SE-I-MSDE, ED-STII, Antananarivo, Madagascar
[2] Thesis director and Laboratory Manager, SE-I-MSDE, ED-STII, Antananarivo, Madagascar
[3] Thesis co-director, SE-I-MSDE, ED-STII, Antananarivo, Madagascar

ABSTRACT

MapReduce framework has become an attractive model to analyze a very large-scale data. One of the techniques that framework is join algorithm. Join algorithm is a complex operation and quite expensive and require sophisticated techniques. In this paper, we present strategies for two-way join relations in a MapReduce environment and study their extension with Bloom Filter approaches. The aim of this work is to show that filters can eliminate the non-joining data as early as possible in order to reduce the costs of disk I/O, communication network and CPU.

Keyword: - *Big data, MapReduce, join algorithms, Bloom filter, Intersection Bloom Filter.*

1. INTRODUCTION

Big data analysis or large-scale data analysis plays an important role in many disciplines especially in business decision making activities. Thanks to e-commerce, social network, search engine, etc. of architecture generate a large data composed of millions of servers. The information generated by its servers is measured in the form of information that is considered to be accurate.

The MapReduce programming model [1] has become popular for big data processing and analyzing large datasets in parallel computing. Its success comes from facilities the process of large datasets in a reasonable amount of time using a large cluster of commodity machines and hiding the details of parallelization, fault tolerance, and load balancing in a simple programming interface. However, MapReduce has severe limitations to performing a join operation with multiple input datasets. To join multiple datasets in MapReduce, all input records have to be sent from map workers to reduce workers, regardless of the size of the joined records.

In the classical context of relational databases, early elimination of useless data is a quite effective technique to reduce the costs of the disk I/O, CPU and communication network of data processing algorithms. So we provide a systematic study of joins with filters for early removal of non-participating tuples from the input datasets. The, we compares the benefit by introducing filters in join algorithms in order to clarify the stakes to combine several entries in MapReduce.

The remainder of this paper is organized as follows: Section 2 summarizes the background and related work, with special focus on MapReduce framework and join based Bloom filter. Bloom join and Join with Intersection Bloom Filter were introduced in section 3. Section 4 describes the cost analysis for two way join. Experimental evaluation will be described in Section 5. Section 6 concludes this paper.

2. BACKGROUND AND RELATED WORK

2.1 MapReduce

MapReduce [1] is a programming model suitable for mass and parallel processing of large amounts of data, executed on a large cluster of commodity machines and highly scalable on thousands of nodes (or machines). The MapReduce architecture consists of a "JobTracker" service that can run on the "NameNode" server and receives "Jobs" jobs from the client applications, and then sends tasks to the "TaskTracker" data nodes available. It allows users to focus on their data operations without worrying about the implementation of features for parallel and distributed processing.

A MapReduce program consists of two distinct functions: map and reduce. The map function takes a input key/value pairs (k_1, v_1) from a Distributed File System (or DFS) and produces a set of intermediate key/value pairs *list*(k_2, v_2). The values in these intermediate pairs associated with the same key k_2 are automatically grouped by the framework and passed to the reduce function. The reduce function aggregates the values to produces final output key/value pairs *list*(k_3, v_3). An execution overview of MapReduce is shown in Figure 1.

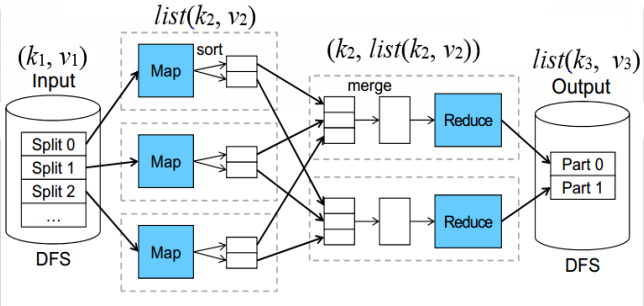

Fig -1: Execution overview of MapReduce

2.2 Bloom Filter

A Bloom (*BF*) filter [2] is a probabilistic data structure used to test whether an element is a member of a set. It consists of an array of m bits and k independent hash functions. All the bits in the array are initially set to 0. When an element is inserted in the array, the element is hashed k times with k hash functions and the positions in the array corresponding to the hash values are set to 1. For To test the membership status of an element, we evaluate this array. If all the bits of the k hash positions of the element are 1, we can conclude that it is in the set. Bloom filters can generate false positives, but false positives are not produced.

A variant of a Bloom filter is *Intersection Bloom filter* [3], denoted *IBF*($S1 \cap S2$), is a probabilistic data structure to check membership in the intersection of sets $S1$ and $S2$. Then, the *IBF*($S1 \cap S2$), is formed by crossing *BF*($S1$) and BF ($S2$) with the bit operator AND. The false positive probability of the intersection filter is estimated as f_{Int} representing one of the probabilities of two approaches to design the filter [3].

2.3 Join algorithms in MapReduce

The processing of joins in MapReduce has become a very interesting topic of research since its introduction by Google in 2004 [4,5,6,7,8]. Several studies have been conducted to analyze join requests for large datasets in a MapReduce environment. So, many implementations on joins have appeared. However, the relative performance of different algorithms depends on several parameters such as input size, data constraints, and so on Map-side join [5,9,10] works by joining two sets of data on the map side without the shuffle and reduce phases. So this algorithm requires some conditions on the input data sets. Each set of input data must be divided into the same number of partitions, sorted by the join key, and must have the same set of keys. All tuples associated with the same key must reside in the same artition in each dataset. In addition, Reduce-sideJoin [5,9,10,11] joins are more general than map-side join for handling a join operation because input datasets are not subject to conditions. Only they incur additional costs because both sets of data must go through the shuffle step in MapReduce.

By analyzing Reduce-side join, we find that many intermediate pairs generated during the Map phase do not participate in the join process due to the lack of correspondence with the pairs of another input dataset. Therefore, eliminating non-matching data directly in the map phase is much more efficient. The semi-join [5] solves this problem by using a distributed cache to broadcast a hash table of one of the input datasets on all mappers, and deleting tuples whose join key is not in the hash table.

However, if the hash table becomes very large, we may overload the memory and replication on all mappers may be inefficient. To overcome this overhead in memory, Bloomjoin [12,8,13,14,15] is the solution because it benefits from a Bloom filter [16] to perform existence tests in less memory than the list complete keys from the hashmap. However, there is still a lot of mismatched data after filtering because the solutions can only filter on one of the input datasets. Thus, the join based on the Intersection filter [3] could become a better solution to solve this problem by eliminating mismatched data from the two input datasets.

3. FILTERING TWO WAY JOINS IN MAP REDUCE

Consider two relations U_1 and U_2 of two-way join $U_1 \bowtie U_2$. Let u_1 (resp., u_2) be the tuples of U_1 (resp U_2) and k represent the join key attribute.

3.1 BloomJoin

Bloomjoin (*BJ*) [12,8,14] is a type of join strategy based on the Bloom filter [9]. BJ is implemented by two MapReduce tasks as follows:

• *Job 1* (preprocessing) is a job with only one reducer. The mappers analyze the splits in the U_2 input, which then extract the value of the join key from each tuple and produce local Bloom filters. Local filters are passed to the reducer by the mappers to merge them into a global filter $BF(U_2)$ using the OR in bits.

• *Job 2* (processing) eliminates the mismatched n-tuples in U_1 and joins the filtered result U_1 to U_2. It uses a distributed cache to store $BF(U_2)$. The mappers analyze the divisions of U_1 and U_2 and eliminate the tuples of U_1 whose keys are not in $BF(U_2)$. U_2 tuples are not filtered.

Each tuple is then checked with a tag indicating its dataset name. In our example, the mappers issue tuples marked with composite keys of the form $((u_1.k, 'U_1'), u_1)$ or $((u_2.k, 'U_2'), u_2)$. The gearboxes receive labeled tuples grouped on the value k (this requires a slight change in the partitioning function). For each group, the reduce function builds all the pairs (u_1, u_2) to complete the join.

Note that it is necessary to override the default grouping function to ensure that the grouping of tagged tuples takes into account only the join key part and ignores the tag portion. The tag is used for secondary sorting that ensures that for a given key value, all tuples in U_1 are processed before those in U_2. This makes it possible to apply a hash join to standard memory.

One of the major problems with the filtering approach in general is the need to perform preprocessing work for filter construction. In addition, the diffusion of the filter becomes ineffective if its size is large. Finally, it should be noted that the *BJ* is asymmetric: the unpaired n-tuples of U_2 have not been filtered, so the problem is half solved. However, *BJ* takes advantage of the compactness of the Bloom filter to reduce the amount of data transferred over the network. In addition, the size of the filter can be defined independently of the number of join keys. But, with a fixed filter size, the probability f of false positives increases with the number of join keys.

3.2 Join based Intersection Bloom Filter

We now describe an improvement of the above approach, the join based intersection Bloom filter [3], denoted *JIBF*. It relies on the fact that only tuples whose join keys belong to the set of shared join keys participate in the result.

The implementation of the *JIBF* is carried out with the following works:

• *Job 1* (filtering) is a job with only one reducer. The groups of U_1 and U_2 are analyzed by the mappers, which then extract the value of the join key for each tuple and insert them into the local Bloom filters, regardless of the duplicate

keys. Local filters are passed to the reducer by the mappers to merge them into two global filters $BF(U_1)$ and $BF(U_2)$ using the OR in bits. And from one of the two approaches to construct the intersection Bloom filter [3], the reducer calculates the *IBF* intersection filter (U_1, U_2) from the global filters.

• *Job 2* (join) uses a distributed cache to provide *IBF* to all the compute nodes. The mappers analyze the divisions of U_1 and U_2, extract the join key for each tuple, and compare it to the intersection filter. If the key v belongs to the intersection filter, the tuple is issued as a pair ((v, tag), tuple). The evaluation of the join in the reduction phase is similar to the algorithm of the Bloomjoin.

JIBF benefits from the standard functionality of Bloom filters: its small size, its independence from the number of keys and their duplication, as well as the quick membership test. The join based on the intersection filter should be more efficient than the Bloomjoin because of its ability to filter out the mismatched n-tuples of the two input datasets. An interesting feature of the intersection filter is that if *IBF* (U_1, U_2) has all the null bits, the sets $U_1.k$ and $U_2.k$ are disjoint and the evaluation of the join stops without anything make. However, the algorithm must pay the additional cost of a MapReduce job for creating the intersection filter and requires the two input datasets to be scanned twice.

4. COST ANALYSIS FOR TWO WAY JOIN

We note *U* and *V* the two input datasets, and analyze the cost for, respectively, the Bloom join (*BJ*) and the join based intersection Bloom filter (*JIBF*). Table 1 summarizes the parameters of our cost model.

4.1 Join with Intersection Bloom Filter

We adapt the cost model presented in [15]. We propose the following global formula that captures the cost of a two-way join.

$$C = C_{read} + C_{filter} + C_{sort} + C_{tr} + C_{write} \quad (1)$$

where

$$C_{read} = c_r.|U| + c_r.|V| \quad (2)$$

$$C_{filter} = 2.c_t.m.r.t \quad (3)$$

$$C_{tr} = c_t.|D| \quad (4)$$

$$C_{sort} = c_l.|D|.2.(\lceil log_B|D| - log_B(m_t)\rceil + \lceil log_B(m_t)\rceil) \quad (5)$$

$$C_{write} = c_r.|O| \quad (6)$$

In equation (1), we add the cost C_{write} and C_{filter} to the cost model described in [17]. C_{write} is the cost of writing the final results. C_{filter} is a constant because the Map Reduce framework parameters, such as the size of a Bloom filter *m*, the number of reduce tasks *r*, and the number of tasktrackers *t*, are set. The coefficient is multiplied by two because local and intersection filters are transmitted between the jobtracker and the tasktrackers. If Bloom filters are not used, C_{filter} is zero. | D |, the size of the intermediate data determines the total cost of the join operation. Thus, we will analyze this parameter to decide the importance of the variant of the algorithm based on the filter.

Table -1: Parameters of the cost model for two-way joins

Parameter	Explanation
$\|U\|$	The size of *U*
$\|V\|$	The size of *V*
$\|D\|$	The size of the intermediate data
c_r	The cost of reading/writing data remotely
c_t	The cost of transferring data from one node to another
m	The compressed size of the Bloom filter (bits) *m* = the size of the Bloom filter × the file compression ratio
r	The number of reduce tasks
t	The number of tasktrackers
c_l	The cost of reading or writing data locally
m_t	The total number of map tasks
$B + 1$	The size of the sort buffer in pages
$\|O\|$	The size of the join processing output

C_{read}	The total cost to read the data
C_{filter}	The total cost to perform the filtering job
C_{tr}	The total cost to transfer intermediate data among the nodes
C_{sort}	The total cost to perform the sorting and copying at the map and reduce nodes
C_{write}	The total cost to write the data on DFS

4.2 Cost Comparaison

We evaluate | D |, for each algorithm mentioned in Sect. 3 and compare the costs. Importantly, we identify a threshold that can guide the choice among these algorithms. We add the reduction side join (RSJ) to our comparison to highlight the effect of filtering.

We denote as δ_U and δ_V, respectively, the ratio of the joined records of *U* with *V* (resp. *V* with *U*). The size of intermediate data is:

$$|D| = \begin{cases} \partial_V|U| + f_{Int}.(1-\partial_V)|U| + \partial_U|V| f_{Int}.(1-\partial_U)|V| & (7) \\ \partial_V|U| + f(V).(1-\partial_V)|U| + |V| & (8) \\ |U| + |V| & (9) \end{cases}$$

where
equation (7) for *JIBF*, denoted D_{JIBF},
equation (8) for *BJ*, denoted $D_{BJ,}$
equation (9) for *RSJ*, denoted D_{RSJ},
f_{Int} (*U, V*) is the false positive probability of the intersection filter *IBF*(*U, V*) [3],
and *f*(*V*) is the false positive probability of the Bloom filter *BF*(*V*).

We can deduce the following important evaluation from these equations.

THEOREM 1. *An JIBF is more efficient than a BJ because it produces less intermediate data. Additionally, the following inequality holds:*

$$|D|_{JIBF} < |D|_{BJ} < |D|_{RSJ} \quad (10)$$

where
D_{JIBF}, D_{BJ}, and D_{RSJ} are the sizes of intermediate data of *JIBF*, *BJ*, and *RSJ*, resp.

PROOF. we get $0 < f_{Int}$ (*U, V*) < f_{Int} (*V*) < 1. We can therefore deduce

$$\partial_U|U| f_{int}(U,V).(1-\partial_V)|U| < \partial_V|U| + f(V).(1-\partial_V)|U| \leq |U| \quad (11)$$
$$\partial_U|V| + f_{Int}(U,V).(1-\partial_U)|V| \leq |V| \quad (12)$$

Combining inequalities (11) and (12) into equations (7), (8) and (9), Theorem 1 is proved to be true.
From Eqs. (1) and (10), we can evaluate the total cost of the join operation for the different approaches.

THEOREM 2. The C_{filter} filtering cost is negligible or less than the cost of non-matching data, then an *JIBF* has the lowest cost. Also, we can derive a cost comparison for the total cost of the different approaches:

$$|C|_{JIBF} < |C|_{BJ} < |C|_{RSJ} \quad (12)$$

where C_{JIBF}, C_{BJ}, and C_{RSJ} are the total costs of *JIBF*, *BJ*, and *RSJ*, resp.

We can therefore deduce that the most efficient join approach is usually *JIBF*, the second is *BJ* and *RSJ* gives the poor performance.

For data localization optimization, the MapReduce framework executes the map task on a node where the input data resides in the DFS and the data is retrieved directly. Thus, the cost of reading this phase is low. As a result, the total C_{filter} cost is negligible compared to the creation and transfer of redundant data on the network.

But, the intersection of filters in a join algorithm will become ineffective when there are a large number of map tasks (m_t), and very little redundant data in the join operation. In the case of so many map tasks, a tasktracker running multiple map tasks will merge the local filters for each task and retain only two local filters *BF*(*U*) and *BF*(*V*). In the case of small redundant data, we will not need to use the filtering job. It is for this reason that we need to estimate the redundant data threshold so that the cost of the filtering job is less than the cost associated with the redundant data and the intersection of the filters becomes more useful.

Let |*D**| the size of the redundant or deleted data, C^*either outputs the total cost to sort and copy the redundant data on the map and reduce nodes, and $C_{tr}^*$is the total cost to transfer redundant data between them nodes. As a result, the cost associated with redundant data is the sum of C_{sort}^* and C_{tr}^*.

THEOREM 3. The filter-based joins become a good choice when.

$$C_{filter} < C_{sort}^* + C_{tr}^*$$

Where

|*D**|=|*U*| + |*V*| - |*D*|,

$C_{tr}^* = c_t \,.|D^*|$,

$C_{sort}^* = c_l.|D^*|.2.(\lceil log_B|D^*| - log_B(mt)\rceil + \lceil log_B(mt)\rceil)$ [17],

c_l: the cost of reading or writing data locally.

Based on the size of the intermediate data |*D*|, the threshold depends on ∂*V* (the recorded joint ratio of *U* with *V*) and ∂*U* (the registered join ratio of *V* with *U*).

5. EXPERIMENTAL EVALUATION

In this section, we present experimental results of our implementation. All experiments were run on a cluster of 5 machines that consists of 1 jobtracker and 4 tasktrackers. Each machine has 3.1 GHz quad-core CPU, 4GB memory, and 100 GB SATA hard disk. The operating system is 64-bit Ubuntu 16.04, and the java version we used is 1.8.0.

We implement the proposed architecture on Hadoop 3.0.1. The HDFS block size was set to 128MB. Each tasktracker can simultaneously run 2 map tasks and 2 reduce tasks. The I/O buffer is set to 128KB, and the memory for sorting data is set to 200MB.

5.1 Datasets

We use a data generation script of the Purdue MapReduce Benchmarks Suite [18], called "PUMA" to produce all test datasets. PUMA represents a broad range of MapReduce applications exhibiting application characteristics with high/low computation and high/low shuffle volumes. The maximum number of columns in the datasets is 25 and string length in each column is set 15 characters. The dataset *dataset1* contains the first column as a foreign key that refers to the fifth column of the dataset *dataset2*. Table 2 summarizes the various dataset sizes used in our experiments.

Table -2: Input datasets

Inputs	Test 1		Test 2		Test 3	
	size	records	size	records	size	records
dataset1	10GB	26 839 442	30GB	79 441 142	50GB	131 908 690
dataset2	10GB	26 738 810	30GB	79 306 710	50GB	126 885 293
Total	20GB	53 578 252	60GB	158 747 852	100GB	258 793 983

For the three test datasets, such as Test 1, Test 2, and Test 3 that we use, we include two *dataset1* and *dataset1*entries. These tests have different sizes, namely 20 GB, 60 GB and 100 GB. All datasets are saved in the same text file format.

The following join query is performed on the dataset.

```
SELECT *
FROM dataset1(c0..c20) d1, dataset2(c0..c20) d2
WHERE          d1.column0 = d2.column5 AND
```

```
d1.ROWNUM <= $number1 AND
d2.ROWNUM <= $number2
ORDER BY        d1.column0
```

The query is executed by changing $*number1* and $*number2* to the number of records of the *dataset1* and the *dataset2*, respectively. An output tuple of the experiments *t* is defined by the concatenation of the pair of tuples of the first 21 columns that joined to produce the output.

5.1 Evaluation

In order to execute the Bloomjoin and the Join based Intersection Bloom Filter algorithms efficiently, we specified the size of filters according to the cardinality of the join key values of datasets and chose the largest filter. There is a tradeoff between *m* and the probability of a false positive. Hence, the probability of a false positive *f* is approximated by:

$$f \approx \left(1 - e^{-\frac{kn}{m}}\right)^k$$

For a given false positive probability *f*, the size of the Bloom filter *m* is proportional to the number of elements *n* in the filter as shown in Table 3.

Table -3: Parameters of filters used in experiments

Tests	Test 1	Test 2	Test 3
f	0.001	0.0001	0.0001
k	7	8	8
n	9910	10526	10526
m/n	15	21	21
m (bit)	148 650	221 046	221 046

where *m*/*n* is the number of bits allocated for each join key and *k* is the number of hash function.

In [19] the parameters optimized for the filter (e.g. *f*, ρ and *m*) can be determined. However, in practice, it is better to choose values less than an optimized value to reduce computational overhead. As shown in Table 3, the values of of *f*, ρ and *m*/*n* in the experiments we conducted were deliberately chosen to determine if they could affect the performance of the join. The filter files generated in the tests are compressed with gzip.

Table 4 gives the intermediate data size (Map output).

Table -4: Number of intermediate tuples (Map output)

Tests	Test 1	Test 2	Test 3
IFBJ	28 969	90 956	162 810
BJ	26 851 277	79 497 558	132 005 845
RSJ	53 547 122	158 655 481	258 645 898

In table 4, it was found that performing a single task does not bring any benefit to the Reduce-side join so it is the most inefficient solution. This is correlated to the large size of intermediate data. Also the number of intermediate tuples generated in this case is almost equal to the number of Map input records, see Tables 2 and 4.

Filter-based joins are more efficient in general. *BJ* and *JIBF* include the preprocessing job and the filtering operation to improve the join performance.

The number of intermediate tuples produced by *BJ* is considerably reduced with respect to *RSJ*. However, in comparison to *JIBF* (see in Table 4), *BJ* still produces much more intermediate data because the filtering operation is only executed on one input dataset (*dataset1*). This situation is overcome by *JIBF*.

Looking at *BJ* and *JIBF*, Table 4 points out that *BJ* generates more intermediate data than *JIBF*. Namely, for the 100 GB test, *BJ* produces 132 005 845 intermediate tuples, whereas *JIBF* produces 162 810 tuples. The experiments reported above are consistent with our theoretical analysis (Theorem 1);

Then, we evaluate the efficiency of these join algorithms by comparing the total execution time. In general fact, the join algorithms generate less intermediate data turn out to be faster, even if we sum up the cost of the preprocessing and join jobs.

Table 4 gives the total execution time of the filtering job and the join job for each algorithm. Regarding filtering, the cost of the filter-based joins is related to the size of the data accessed to build the filter(s). In particular, *JIBF* has to scan two input datasets. However, it pays off, since once the filters are available, the cost of join jobs is drastically reduced.

Table -4: Execution of filtering job and join job (in minutes)

Join algo.	Test 1 (20 GB)			Test 2 (60 GB)			Test 3 (100 GB)		
	Filtering job	Join job	Total time	Filtering job	Join job	Total time	Filtering job	Join job	Total time
IFBJ	8.51	17.55	26.06	16.40	60.72	77.12	28.20	227.62	255.82
BJ	5.15	44.21	49.36	8.10	114.65	122.75	12.89	345.28	358.17
RSJ	0	74.80	74.80	0	210.41	210.41	0	373.00	373.00

Figure 2 demonstrates that the best execution results from using intersection filters. Their total execution time is significantly reduced compared to *BJ* in spite of the time spent in the filtering job.

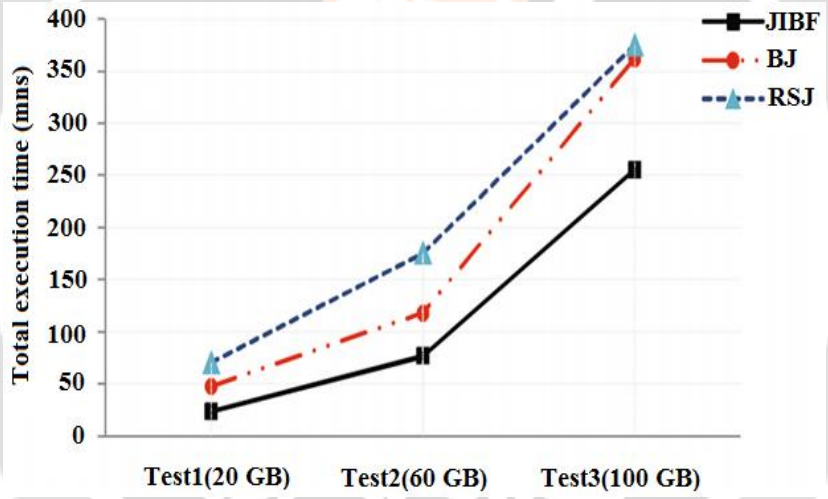

Fig -2: Total execution time

The total execution time of IFBJ increases from about 26.06 to 255.82 (mns), whereas that of BJ ranges from 49.36 to 358.17 (mns). The worst execution is RSJ, ranging 74.80 to 373 (mns). The smaller cost of IFBJ compared to the others (Table 4), is analyzed in Theorem 2.

6. CONCLUSION

In this paper, we have presented join and join based bloom filter algorithms in the MapReduce framework. Filters are known to greatly improve the cost of distributed joins thanks to their ability to avoid network transfer of useless data. We have shown how to adapt join algorithms with filters, systematically. In addition, we model the cost that serves as a benchmark for comparing the expected efficiency of joins. And then evaluate our algorithms on a complete set of experiments to validate our models.

To conclude, join evaluation using filters is more efficient than other solutions since it reduces the need for s hipping non-matching data. Future work will consist to evaluate our methods for multi-way joins.

7. REFERENCES

[1]. J. Dean and S. Ghemawat, "*Mapreduce: Simplified data processing on large clusters*", Commun. ACM, Vol. 51, No. 1, pp. 107–113, Jan. 2008.
[2]. B. H. Bloom. "*Space/time trade-offs in hash coding with allowable errors*", Communications of the ACM (CACM), Vol.13, No. 7, pp. 422–426, 1970.
[3] M. A. T. RAJAONARIVELO, S. RAKOTOMIRAHO, R. M. RANDRIAMAROSON "Improving two-way join processing using bloom filter intersection in map reduce", Vol.4, Issue-6,IJARIIE-ISSN(O)-2395-4396, 2018.
[4]. Afrati, F.N., Borkar, V .R. Carey, M.J., Polyzotis, N ., Ullman, J.D., "*Map-reduce extensions and recursive queries*", In: Proceedings of the International Conference on Extending Database Technology (EDBT), Uppsala, Sweden, pp. 1–8, 2011.
[5]. Blanas, S., Patel, J.M., Ercegovac, V., Rao, J., Shekita, E.J., Tian, Y., "*A comparison of join algorithms for log processing in mapreduce*", In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, pp. 975–986. ACM, New York, 2010.
[6]. Bruno, N., Kwon, Y., Wu, M.C., "*Advanced join strategies for large-scale distributed computation*", Proc. VLDB Endow. **7**(13), pp. 1484–1495, 2014.
[7]. Hassan, M.A.H., Bamha, M., "*Semi-join computation on distributed file systems using map-reduce-merge model*", In: Proceedings of the Symposium on Applied Computing (SAC), Sierre, Switzerland, pp. 406–413, 2010.
[8]. Lee, T., Kim, K., Kim, H.J.: "*Join processing using Bloom filter in MapReduce*", In: Proceedings of the RACS, San Antonio, TX, USA, pp. 100–105, 2012.
[9]. Lee, K.H., Lee, Y.J., Choi, H., Chung, Y.D., Moon, B., "*Parallel data processing with mapreduce: a survey*". SIGMOD Rec. 40(4), pp. 11–20, 2012.
[10]. White, T., "*Hadoop: The Definitive Guide. O'Reilly*", Sebastopol, 2012.
[11]. Liu, L., Y in, J., Gao, L.,"*Efficient social network data query processing on MapReduce*", In: Proceedings of the Workshop on HotPlanet, Hong Kong, China, pp. 27–32, 2013.
[12]. Lam, C., "*Hadoop in Action*", Manning Publications, Greenwich, 2010.
[13]. Lee, T., Kim, K., Kim, H.J., "*Exploiting bloom filters for efficient joins in MapReduce*". Inf. Int. Interdisc. J. 16(8), 5869–5885, 2013.
[14]. Zhang, C., Wu, L., Li, J., "*Optimizing distributed joins with bloom filters using MapReduce*", In: Kim, T., Cho, H., Gervasi, O., Yau, S.S. (eds.) GDC, IESH and CGAG 2012. CCIS, Vol. 351, pp. 88–95. Springer, Heidelberg, 2012.
[15]. Zhang, C., Wu, L., Li, J., "*Efficient processing distributed joins with bloom filter using mapreduce*", Int. J. Grid Distrib. Comput. (IJGDC) **6**(3), pp. 43–58, 2013.
[16]. Bloom, B.H., "*Space/time trade-offs in hash coding with allowable errors*", Commun. ACM 13(7), pp. 422–426, 1970.
[17]. Nykiel, T., Potamias, M., Mishra, C., Kollios, G. and Koudas, N. 2010, "*MRShare: sharing across multiple queries in MapReduce*", Proc. VLDB Endow, Vol. 3, No.1 -2, pp. 494–505, Sept. 2010
[18]. Ahmad, F.: "*Puma benchmarks and dataset downloads* (2012)". https://engineering. purdue.edu/ ~puma/datasets.htm, 2018.
[19]. Broder, A.Z., Mitzenmacher, M., "*Survey: network applications of Bloom filters: a survey*". Internet Math. 1(4), pp. 485–509, 2003.