IMPROVING TWO-WAY JOIN PROCESSING USING BLOOM FILTER INTERSECTION IN MAP REDUCE

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

Big Data Analysis is at the heart of modern enterprise and scientific research. To analyze this very large-scale data, MapReduce has become an attractive model for this kind of data. However, this model is not designed to perform join operations with multiple entries. Although many studies on join algorithms like Bloomjoin in MapReduce have been explored, but a lot of non-joining data is still being generated and transmitted over the network. This research will provide us with a model to solve this problem by making an intersection filter based on the Bloom Filter to eliminate disjoint elements between two sets of input data. Two methods are proposed to model the filter. To apply the filter in a join operation, a MapReduce job will be adjusted consistently, attempting not to increase the associated costs. Therefore, thanks to the intersection filter, the operation of the joins minimizes the cost of disk I/O. Finally, the study carried out proved its effectiveness by comparing with the existing solutions on the costs of the joins of the two approaches.

.

Keyword : *Big data, MapReduce, Bloom filter, joing processing*

1. INTRODUCTION

Big Data or Very large data sets represent a challenge in many discipline. Giant Internet companies as Google, Facebook, etc. want to analyze terabytes of application logs and clickstream data, scientists have to process data sets collected by large-scale experiments and sensors and the governments of useful and personal data.

For the government, all information is essential for developing predictive models in various fields. Big data serves to find solutions to unresolved problems of science for scientist. However for traders, they want to find patterns in customer and sales data to improve their business. But to ensure an answerable time in this analysis, parallel computing is essential. MapReduce [1] has been extensively used for large-scale data analysis in academic and business areas. MapReduce has become popular and the success stems from hiding the details of parallelization, fault tolerance, and load balancing in a simple programming framework.

Unfortunately, MapReduce has some limitations to performing a join operation on multiple datasets because MapReduce does not directly support operations with multiple inputs. Several researches are carried out to solve this problem and gave several solutions [2][3] where a join operation will be compiled to MapReduce job(s). However, join processing in MapReduce is that it emits large intermediate results in the map phase do not actually participate in the joining process, regardless of the number of final join results. A part of the problem can be solved by using a filter called Bloom Filter (BF) [4] which is a space-saving probabilistic structure for testing set membership. BF is built for one of the input datasets and is distributed to all mappers. When the mapper receives

tuples from another data set of inputs, it looks at the join keys and eliminates tuples that do not match in the filter. However, this solution only applies to one of the input data sets to filter the tuples that do not join but not to the two input data sets. It is therefore necessary to find a better solution to improve this situation.

Using the BF in MapReduce, we can see that the result of the join operation only contains tuples whose join keys belong to the intersection of the projected datasets on the join key. This paper introduces a new type of Bloom filter called Intersection Bloom filter, representing the intersection of the datasets for the join operation. For this solution, the disjoint elements are deleted directly during a query for each tuple of the input datasets.

In this article, we bring three main contributions: two approaches to model the intersection of the filter, the optimization of two-way join with its approaches and finally we demonstrate that the intersection of the filter is more efficient than the basic Bloom filter in a join operation.

2. JOIN OPERATION ON BLOOM FILTER

2.1 Bloom Filter

A Bloom (BF) filter [4] is a probabilistic data structure used to test whether an element is a member of a set. It consists of an array of bits indexed by a tuple of independent hash functions. Two configurations of bit indexing are generally used, called unpartitioned and partitioned. The k-tuple hash function of an unpartitioned Bloom filter uniformly maps to the entire m-bit range of the bit-array, $h : S \rightarrow [m]^k$. In contrast, a partitioned Bloom filter distinguishes k disjoint sub-arrays of the filter, with each hash tuple value uniformly mapping to an integer $\frac{m}{k}$ -bit partitioned range, $h : S \rightarrow [\frac{m}{k}]^k$ [5].

A Bloom BF (S) filter to represent a set S of n elements is described as follows:

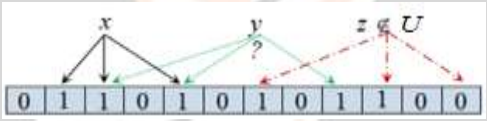

Fig -1: Unpartitioned Bloom filter *BF*(*U*)

- The set $U = \{u_1, u_2, ..., u_n\}$ of *n* elements is represented by an array of *m* bits, initially set to 0.
- The filter uses *k* hash functions independent $H=\{h_1, h_2, ..., h_k\}$ which returns values in the range {1 ... *m*} such that $h_i: X \rightarrow \{1..m\}$, where *X* is the value of the key.
- To insert an element $x \in U$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$. A location can be set to 1 many times, but only the first change has an effect.
- To check if an element $y \in U$, we check if all $h_i(y)$ are set in the bit array. If not, then clearly $y \notin U$ if at least one of these positions is set to 0. If all $h_i(y)$ are set to 1, we assume that *y* is in *U* even though it might be false with some probability.

Fig -2 represents a partitioned Bloom filter. It is defined by an array of *m* bits that is partitioned into *k* disjoint arrays of size $m_p=m/k$ bits.

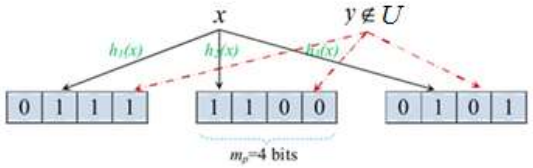

Fig -2: Partitioned Bloom filter *BF*(*U*)

The *BF*(*U*) consists of three 4-bit partitions, *k*=3 hash functions and size *m*=12 bits.

In the *BF*(*U*), we insert an element $x \in U$ by computing $h_i(x)$ and setting the corresponding position in the i^{th} partition to 1 for $1 \leq i \leq k$. Also, we test if the element *y* is in *S* by checking the position corresponding to $h_i(y)$ in the i^{th} partition.

The probability of the unpartitioned filter or basic filter *p* and the partitioned filter p_p is asymptotically equivalent of a bit set to 0. Indeed, the basic filter tends to work slightly better than the partitioned filter, because when $k > 1$, the basic filter tends to have more than 0 as the partitioned filter as shown in the expression as follow:

$$p = \left(1 - \frac{1}{m}\right)^{kn} > p_p = \left(1 - \frac{k}{m}\right)^{n} \quad (1)$$

Note that the partitioned filter is more flexible than the basic filter because we can always change the probability of false positive after creating the filter by increasing or reducing the partitions without resizing. Therefore, the resolution of one of Bloom's filters can be adjusted by eliminating or increasing his partition keeping the same partition size.

2.2 MapReduce

MapReduce [6] is a parallel programming model to simplify large-scale data processing on parallel and distributed architectures. Users only need to provide two functions, called map and reduce, and the framework handles all issues related to parallelization, fault tolerance, data distribution, and load balancing.

In the implementation of the MapReduce program (**Fig -2**), there are two types of processes, the workers, which execute map and reduce tasks and the master, which is responsible of controlling the workers' execution.

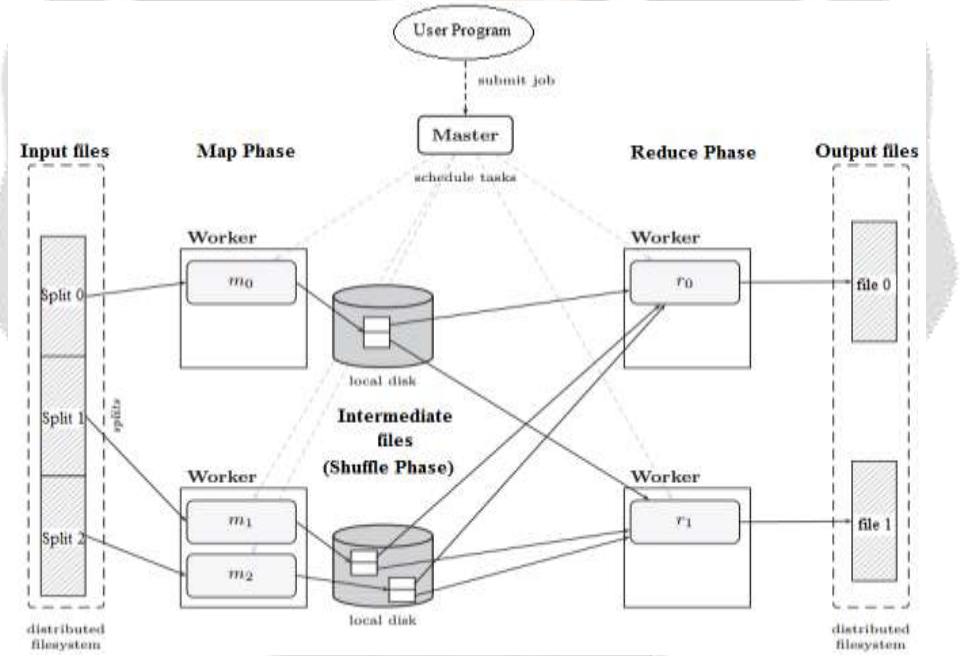

Fig -3: Execution overview of MapReduce

2.3 Join in MapReduce

The join algorithms in MapReduce are generally categorized as map-side join [7] and reduce-side join [7]. For map-side join, we observe many intermediate pairs generated during the map phase that do not participate in the join process due to the lack of match with the pairs of another input dataset. Therefore, it would be much more efficient to eliminate the non-matching data directly in the map phase. This problem can be solved in semi join [8]. It filters the redundant input data before the join process by using a distributed cache to broadcast a hash table of one of the input datasets on all mappers. Another solution to this problem is the broadcast join [9] which is another option if one of the datasets is very small, so that it can match the memory. The main obstacle to these solutions is the hash table, which may not fit in memory and replication on all mappers may be inefficient. Bloomjoin [10] [11] or join algorithm using Bloom filters is an advantageous and original solution for filtering tuples that are not associated with a join, as shown in **Fig -4**.

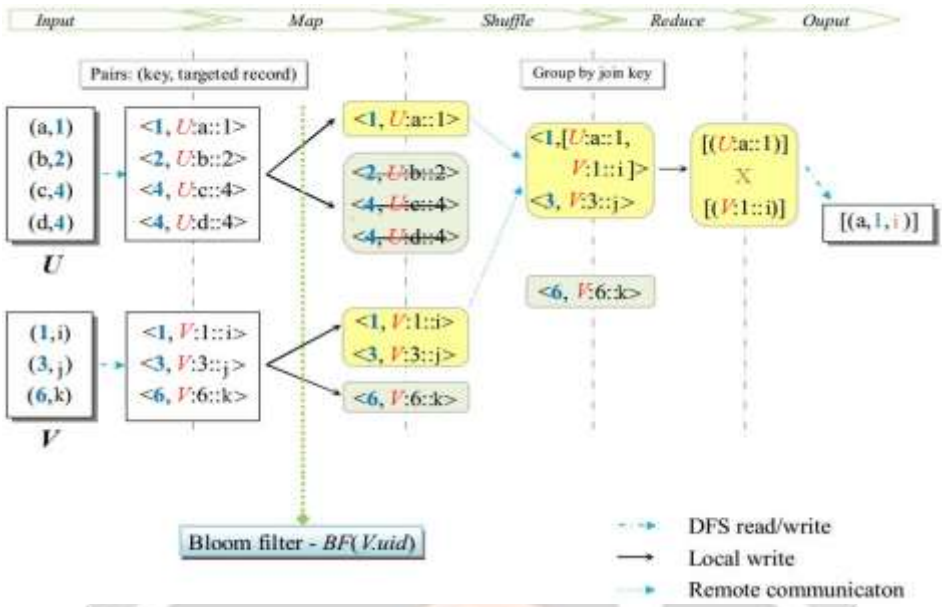

Fig -4: Bloomjoin in MapReduce

3. MODELING INTERSECTION FILTER BASED ON BLOOM FILTER

In Bloomjoin, basic Bloom filters are used. But, these filters only have the ability to remove redundant data from one of the input datasets instead of both; therefore, there remains a large amount of redundant data of another dataset that is passed to join processing. A better replacement for the basic filters is the intersection filter with the ability to filter out disjoint elements between two datasets. We show in this section two approaches to build the intersection filter.

In this paper, a set X can be presented as a bloom filter through a mapping relation: $X \rightarrow BF(X)$. We use two bloom filters $BF(U)$ and $BF(V)$ as the concise representation of two different given sets U and V respectively.

3.1 Approach 1: Intersection of unpartitioned Bloom

The main idea of this approach is that the intersection of the Bloom filters results in a filter called intersection filter.

By arranging the expression of the intersection of the filters in [8], we obtain $BF(U \cap V) = BF(U) \cap BF(V)$ with probability$(1 - 1/m)^{k|U-U\cap V|\times k|V-U\cap V|}$.

However, to calculate the intersection of the filters $BF(U \cap V)$, the intersection of the two filters $BF(U) \cap BF(V)$ is not sufficient. Therefore, we will have to look for an approximation by joining *BF (U)* and *BF (V)* with the bit-wise AND and keep the essential query characteristics [12] [13]. This indicates that in the intersection of the filters, all k hash functions for join key x corresponding to bit 1, then x belongs to $U \cap V$ with high probability.

Fig -5 shows the intersection of filters with two unpartitioned Bloom filters using the same size m and k hash functions.

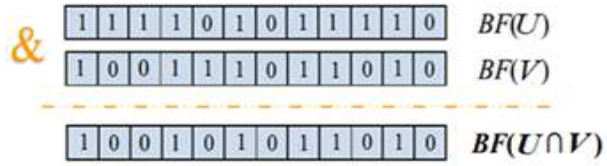

Fig -5: Intersection of unpartitioned Bloom filters

The intersection of the filters BF (U ∩ V) is formed by crossing BF (U) and BF (V) with the bit operator AND.

This approach allows us to use a single intersection Bloom filter to remove most of the non-joined tuples from the two *U* and *V* input datasets.

3.2 Approach 2: Intersection of partitioned Bloom

In this approach, we always keep the same idea of constructing the intersection of filters but using two partitioned Bloom filters.

The size of partitioned Bloom filters can be changed after creation. Therefore, we can adapt the partitioned filters *BF* (*U*), *BF* (*V*) and $BF(U) \cap BF(V)$by decreasing or increasing the partitions without re-hashing to guarantee the same size m and k hash functions.

Fig -6 describes the construction of the intersection of the filters.

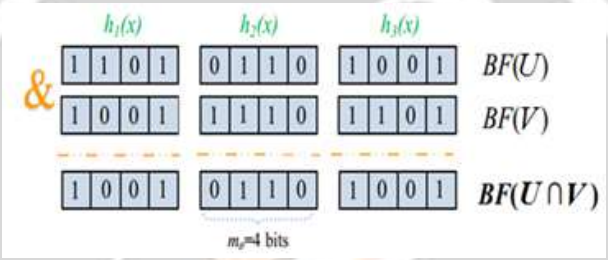

Fig -6: Intersection of partitioned Bloom filters

To create the intersection filters *BF* (*U* ∩ *V*), two partitioned Bloom filters *BF*(*U*) and *BF*(*V*) with 3 partitions are crossed in pairs with the AND bit operator. The result of the operation produces 3 partitions of 4-bit.

We notice that in this approach, the two input data sets are disjoint if there is at least one resulting filter in a partition in which all *m*/*k* bits are equal to 0. Therefore, join processing can be completed without doing anything. This essential feature for joint operations is not present in the first approach which speeds up the processing time.

3.3 False intersection probability

After defining the two types of intersection of Bloom's filters, we now model their probability of false positives. The probability of a false positive for an element not in the set can be calculated as the following expression.

$$f = (1-p)^k = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k \quad (2)$$

Fig -7 shows the intersection of two sets with false positives.

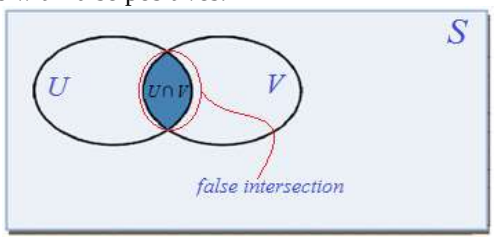

Fig -7: false positive of two sets using Bloom filters

The false positive of the intersection is defined by the bright area inside the red oval while the actual intersection of the two sets of *U* and *V* data are in the blue zone marked *U*∩*V*. It is also called false intersections. Concerning the following theorems, let's fix two sets *U*, *V* ⊂ *S*.

THEOREM 1. *A false positif by Bloom filter intersection of unpartitioned BF(U) and BF(V) is reported with probability*

$$f_{uBF} = \left(1 - \left(1 - \frac{1}{m}\right)^{k|U|}\right)^k \left(1 - \left(1 - \frac{1}{m}\right)^{k|V|}\right)^k \quad (3)$$

where *BF*(*U*), *BF*(*V*) and *BF*(*U* ∩ *V*) have the same size m and k hash functions.

PROOF. The intersection of the unpartitioned Bloom filters causes false intersections when all k bits in the resulting bit array is set to 1 from two different join keys. From equation (2), the probability for *k* bits to be set in both *BF*(*U*) and *BF*(*V*) from two different keys is the product of f_U and f_V.
where

$$f_U = \left(1 - \left(1 - \frac{1}{m}\right)^{k|U|}\right)^k \quad (4)$$

and

$$f_V = \left(1 - \left(1 - \frac{1}{m}\right)^{k|V|}\right)^k \quad (5)$$

THEOREM 2. *A false positif by Bloom filter intersection of partitioned BF(U) and BF(V) is reported with probability*

$$f_{pBF} = \left(1 - \left(1 - \frac{k}{m}\right)^{|U|}\right)^k \left(1 - \left(1 - \frac{k}{m}\right)^{|V|}\right)^k \quad (6)$$

where *BF*(*U*), *BF*(*V*) and *BF*(*U* ∩ *V*) have the same size m and k hash functions. k partitions are the same size m_p = *m* / *k*.

PROOF. Like to Theorem 1, the probability for *k* bits to be set in *k* partitions of *BF*(*U*) and *BF*(*V*) with two different keys is the product of f_U and f_V.
where

$$f_U = \left(1 - \left(1 - \frac{k}{m}\right)^{|U|}\right)^k \quad (7)$$

and

$$f_V = \left(1 - \left(1 - \frac{k}{m}\right)^{|V|}\right)^k \quad (8)$$

THEOREM 3. *The false intersection probability of the partitioned filter intersection is more than the false intersection probability of the unpartitioned filter intersection* $f_{pBF} > f_{uBF}$

PROOF. If Bloom filters have more bits set to 1, the probability for a bit collision can be higher. And thus partitioned filters tend to have more 1's than unpartitioned filters.
As equation (1), we get

$$1 - p_p = 1 - \left(1 - \frac{k}{m}\right)^{|U|} > 1 - p = 1 - \left(1 - \frac{1}{m}\right)^{k|U|}$$
$$\left(1 - p_p = 1 - \left(1 - \frac{k}{m}\right)^{|U|}\right)^k > \left(1 - p = 1 - \left(1 - \frac{1}{m}\right)^{k|U|}\right)^k \quad (9)$$

where

$$f_{pBF} > f_{uBF}$$

4. Optimization for two-way joins using intersection filters in MapReduce

4.1 Execution Overview

We implement a 2-way join of the two arbitrary input datasets. We will use the *Reduce-side join* algorithm and evaluate the join operation in general. Nevertheless, we can still use our intersection of filters for the other join algorithms.

We have implemented our approach into the Hadoop [14] wich is the main platform of Big data, an open-source implementation of the MapReduce framework. In Hadoop, the master node is called the jobtracker and the worker node is called the tasktracker.

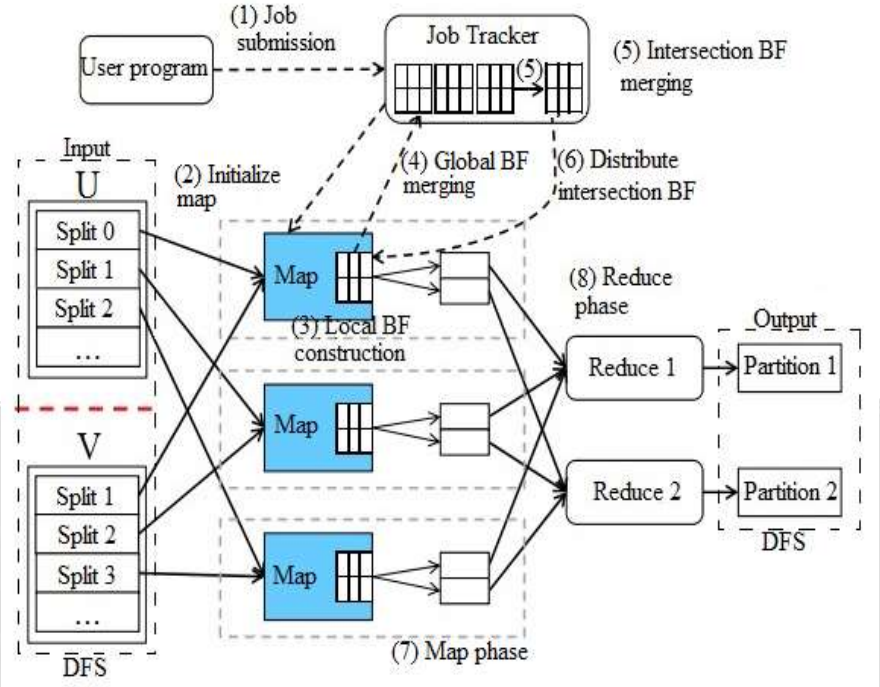

Fig -8. Two-way join execution with intersection filter in MapReduce

When the user program is submitted, the following sequence of actions is performed in our join operation with the support of our intersection filter.

1. **Job submission.** A join job together with its input datasets *U* and *V* is configured and submitted by a client to the jobtracker. m_{t1} map tasks for *U*, m_{t2} map tasks for *V*, and *r* reduce tasks are created.

2. **Map phase.** The jobtracker assigns each split of the m_{t1} map tasks for *U*, m_{t2} map tasks for *V* or the *r* reduce tasks to idle tasktrackers. A map tasktracker reads each tuple from its split, converts it to a <key, tuple> pair, and then call the map function for the input pairs.

3. **Local BF construction.** The intermediate pairs produced from the map function are divided into *r* partitions, which are sent to *r* tasktrackers respectively. Also, *r* Bloom filters are constructed on the keys in each partition i.e. *BF*(*U*) for the dataset *U* and *BF*(*V*) for the dataset *V*. These filters are called local filters because they are built for only the intermediate results in memory on tasktrackers.

4. **Global BF merging.** Once all map tasks are complete, the jobtracker signals all tasktrackers and they send their local filters via heartbeat responses to the jobtracker. Then, the jobtracker merges the local Bloom filters to constructs two global filters *BF*(*U*) and *BF*(*V*).

5. **Intersection BF merging.** It computes and builds the intersection filter $BF(U \cap V)$ based on our proposals from the two global filters. If the intersection filter is empty, the entire join job will be finished and the jobtracker responds with instructions to stop tasks or the job.

6. **Distribute intersection BF.** If the intersection filter exist, the intersection filter is distributed to all nodes in a cluster using a distributed cache.

7. **Map phase.** The map function queries the join key of the tuple into the intersection filter *BF* ($U \cap V$). If it exists, then the map has a tagged pair <dataset-name :: join-key, tuple> and sends this pair to a corresponding reducer. The input key/value pairs which are not set in intersection filter are filtered out.

8. **Reduce phase.** The reduce function will take its input and do a full crossproduct of tuples of different input datasets for a given join key to get our joined output. Final output results are written into Distributed File System (DFS).

4.2 Merging Bloom Filters

The merger of Bloom's filters is easier to implement than the intersection of Bloom's filters. The merge operation corresponds to the construction of the union filter *BF* ($U \cup V$). We can build the union of Bloom filter as shown in the Theorem 4.

THEOREM 4. *If* $BF(U_1 \cup U_2)$, $BF(U_1)$ *and* $BF(U_2)$ *use the same m and hash functions, then* $BF(U_1 \cup U_2) = BF(U_1) \cup BF(U_2)$.

PROOF. Assume that the number of hash functions is *k*. We choose an element *y* from set $U_1 \cup U_2$ randomly, and *y* must also belong to set U_1 or U_2. Bits $h_i(y)$ of $BF(U_1 \cup U_2)$ are set to 1 for $1 \le i \le k$, and at the same time, bits $h_i(y)$ of $BF(U_1)$ or $BF(U_2)$ are set to 1, thus $BF(U_1)[h_i(y)] \cup BF(U_2)[h_i(y)]$ are also set to 1. Instead, we chose an element *x* from set U_1 or U_2 randomly, and *x* also belong to set $U_1 \cup U_2$. Bits $h_i(x)$ of $BF(U_1) \cup BF(U_2)$ are set to 1 for $1 \le i \le k$, and at the same time, bits $h_i(x)$ of $BF(U_1 \cup U_2)$ are also set to 1. Thus, $BF(U_1 \cup U_2)[i] = BF(U_1)[i] \cup BF(U_2)[i]$ for $1 \le i \le m$.

We can actually extend Theorem 4 out to the following fact.

THEOREM 5. *If* $BF(U_1)$, $BF(U_2)$,..., $BF(U_n)$ *and* $BF(U_1 \cup U_2 \ldots \cup U_n)$ *use the same size m and k hash functions, then* $BF(U_n)$ *and* $BF(U_1 \cup U_2 \ldots \cup U_n) = BF(U_1) \cup BF(U_2) \ldots BF(U_n)$.

The union of Bloom filters with the same size and set of hash functions is implemented by bit-wise OR.
In the Global BF merging step, the jobtracker can collect all local Bloom filters from tasktrackers, merge the filters by using the bit-wise OR of two bit-arrays, and generate the global Bloom filter. And in Intersection BF merging, the global filter is intersected with another global filter to create the intersection filter $BF(U \cap V)$.

5. COST ANALYSIS

5.1 Cost Model

A cost model for MapReduce is presented in [15]. We will adapt this model to the model of our implementation. Assume that we have a MapReduce job that read the input dataset *U* and *V*. Let $|U|$ be the size of *U*, $|V|$ be the size of *V*, and $|D|$ the size of the intermediate data of the join job. We also assume that $c_l$is the cost to read or write data locally, c_t is the cost to transfer data from one node to another, c_r is the cost of reading or writing data remotely, $|O|$ is the size of the ouput data, *t* is the number of tasktrackers, and the size of the sort buffer is $B + 1$.

The total cost for a join operation between *U* and *V* is the sum of the total cost $C_{read}$to read the data, the total cost C_{filter} to transfer Bloom filters among the nodes (corresponds to step (3) to (6) in **Fig -8**), the total cost $C_{write}$to write the data, the total cost $C_{sort}$to perform the sorting and copying at the map and reduce nodes, the total cost C_{tr}totransfer intermediate data among the nodes. Then, the total cost *C* is as follows:

$$C = C_{read} + C_{filter} + C_{sort} + C_{tr} + C_{write} \quad (10)$$

where

$$C_{read} = c_r.|U| + c_r.|V| \quad (11)$$

$$C_{filter} = 2.c_t.m.r.t \quad (12)$$

$$C_{tr} = c_t.|D| \quad (13)$$

$$C_{sort} = c_l.|D|.2.(\lceil log_B|D| - log_B(m_{t1} + m_{t2})\rceil + \lceil log_B(m_{t1} + m_{t2})\rceil) \quad (14)$$

$$C_{write} = c_r.|O| \quad (15)$$

In equation (10), we add the cost $C_{write}$and C_{filter} to the cost model described in [15]. $C_{write}$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. An additional difference in our model is the size of intermediate results |*D*| is changed according to the performance of Bloom filters. This is discussed in the following subsection.

5.2 Cost comparison of approaches

The important factor that determines the total cost is the size of intermediate results |*D*|. Let $\sigma_U$is the ratio of the joined records of *V* with *U*, and σ_V is the ratio of the joined records of *U* with *V*. Then we can estimate |*D*| with the false intersection of probability as follows:

$$|D| = \begin{cases} \partial_V|U| + f_{uBF}.(1-\partial_V)|U| + \partial_U|V| f_{uBF}.(1-\partial_U)|V| & (16) \\ \partial_V|U| + f_{pBF}.(1-\partial_V)|U| + \partial_U|V| f_{pBF}.(1-\partial_U)|V| & (17) \\ |U| + |V| & (18) \end{cases}$$

where
equation (16) for the unpartitioned intersection filter (approach 1),
equation (17) for the partitioned intersection filter (approach 2),
equation (18) if Bloom filter is not used,
and f_{uBF} et f_{pBF} refer to Section 3.3.

We can learn from this equation of intermediate results |*D*|, the following evaluation imports.

THEOREM 6. *The join operation using a basic Bloom filter is less efficient than using an intersection filter because it produces a lot of redundant and intermediate data than the intersection filter.*

$$|D|_{16} < |D|_{17} < |D|_{18} \quad (19)$$

where $|D|_i$ is the intermediate data size for equation i^{th} (i = 16, 17, 18).

PROOF. From Theorem 3, we get $0 < f_{uBF} < f_{pBF} << 1$. We can therefore deduce

$$\partial_U|V| f_{uBF}.(1-\partial_U)|V| < \partial_U|V| + f_{pBF}.(1-\partial_U)|V| < |V| \quad (20)$$

$$\partial_V|U| f_{uBF}.(1-\partial_V)|U| < \partial_V|U| + f_{pBF}.(1-\partial_V)|U| < |U| \quad (21)$$

Combining inequalities (20) and (21) into equations (16), (17) and (18), Theorem 6 is proved to be true.

From equations (10) and (19), we can evaluate the total cost of the join operation for the different approaches by the following theorem.

THEOREM 7. *The cost of join operation using the intersection filter is the lowest. Also, we can derive comparing equation for the total cost of the algorithms:*

$$|C|_{16} < |C|_{17} < |C|_{18}$$

where C_i is the total cost in case of equation i^{th} (i = 16, 17, 18).

6. CONCLUSION

In this paper, we have presented an architecture to improve join performance of two-way join using intersection of Bloom filters in the MapReduce framework. We have made two approaches for building the intersection filter. First, we use unpartitionned Bloom Filter or partitioned filter to build the intersection filter. Second, we have evaluated our approaches comparing the cost of join operation and the cost of intermediate data. The results show that the join operation using the intersection filter is more efficient than other solutions since it significantly reduces redundant data, and thus produces much less intermediate data. In addition, the intersection filter has false positive results and an additional cost for the construction of the filter, its efficiency in terms of space saving and filtering which really compensates for these disadvantages.

To conclude, optimizing two-way join using the intersection filter offers a new solution for researchers in MapReduce framework as well as in the Big Data field. In addition, an experimental evaluation will also be considered.

7. REFERENCES

[1]. D. Jeffrey and G. Sanjay, "*MapReduce: Simplified Data Processing on large Clusters*", in OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, vol. 51, USENIX Association, 2004

[2]. Blanas, S., Patel, J.M., Ercegovac, V., Rao, J., Shekita, E.J. and Tian, Y. 2010. A comparison of join algorithms for log processing in MaPreduce. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, New York, NY, USA, pp. 975–986, 2010

[3]. Vernica, R., Carey, M.J. and Li, C. 2010. Efficient parallel set-similarity joins using MapReduce. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, New York, NY, USA, pp. 495–506, 2010

[4]. 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

[5]. Kirsch, A. and Mitzenmacher, M. 2006. Less hashing, same performance: building a better bloom filter. *Proceedings of the 14th conference on Annual European Symposium Volume 14* (London, UK, UK, 2006), 456–467, 2006

[6]. J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008

[7]. L. Jena,S. S. kumar, S. Santosh and D.S. Gayatri, "*Comparison of Two Way Join Algorithms used in MapReduce Frame work for handling BigData*", in International Journal of New Innovations in Engineering and Technology, vol. 4, no. 2, pp. 16-25, January 2016

[8]. M. A. H. Hassan and M. Bamha, "*Semi-join computation on distributed file systems using map-reduce-merge model*", in Proceedings of the 2010 ACM Symposium on Applied Computing, New York, NY, USA, pp. 406–413, 2010.

[9]. S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "*A comparison of join algorithms for log processing in MaPreduce*" in Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, New York, NY, USA, pp. 975–986, 2010.

[11]. Lee, T., Kim, K. and Kim, H.-J. 2012. Join processing using Bloom filter in MapReduce. *Proceedings of the 2012 ACM Research in Applied Computation Symposium* , New York, NY, USA, pp.100–105, 2012

[10]. S. M. Mahajan and V. P. Jadhav, "BLOOM JOIN FINE-TUNES DISTRIBUTED QUERY IN HADOOP ENVIRONMENT" in International Journal of Advanced Engineering Technology, vol. 4, Issue 1,pp. 67-71, 2013

[12]. Broder, A. and Mitzenmacher, M. 2004. Network Applications of Bloom Filters: A Survey. Internet Mathematics. vol. 1, no. 4, pp. 485–509. 2004

[13]. Guo, D., Wu, J., Chen, H., Yuan, Y. and Luo, X. 2010. The Dynamic Bloom Filters. IEEE Transactions on Knowledge and Data Engineering, vol. 22, no.1, pp. 120 –133, January 2010

[14]. Hadoop. http://hadoop.apac

[15]. 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 , 494–505, September 2010