Join Query Processing in Data Quality Management

Mingliang Yue(✉), Hong Gao, Shengfei Shi, and Hongzhi Wang

School of Computer Science and Technology,
Harbin Institute of Technology, Harbin, China
{ml_yue, honggao, wangzh}@hit.edu.cn

Abstract. Data quality management is the essential problem for information systems. As a basic operation of Data quality management, joins on large-scale data play an important role in document clustering. MapReduce is a programming model which is usually applied to process large-scale data. Many tasks can be implemented under the framework, such as data processing of search engines and machine learning. However, there is no efficient support for join operation in current implementations of MapReduce. In this paper, we present a strategies to build the extend bloom filter for the large dataset using MapReduce. We use the extend bloom filter to improve the performance of two-way and multi-way joins.

Keywords: Data quality management · MapReduce · Bloom filter · Join

1 Introduction

In recent year, with the wide popularity of Internet technology, along with the rapid development of cloud computing technology, the data in the Internet is growing at an unprecedented speed and accumulation. Data quality management [1] is the essential problem for information systems. As a basic operation of Data quality management, joins on large-scale data play an important role in document clustering.

Hadoop [2] provides a default join mechanism for relational data, the MapReduce programming model is widely applied to massive data based processing because of its good scalability, fault tolerance and usability. However because of its own limitation, the performance of MapReduce [3] is slow when it is adopted to perform complex data analysis tasks that require the joining of data sets in order to compute certain aggregates. Therefore, it is necessary to design an improved method for Join operation under MapReduce framework.

Aiming at the shortage for processing join operations based on MapReduce. In this paper we presents a join algorithm based on extend Bloom Filter, whose core idea is to use Bloom Filter to decrease the network overhead between the map and reduce phases so as to improve the efficiency. First of all, an efficient algorithm building a Bloom Filter for a dataset is proposed; secondly, join algorithms based on Bloom Filter axe proposed, including two-way and multi-way. In this paper, we study relational data join within MapReduce, and make the following contributions:

H. Gao et al. (Eds.): DASFAA 2016 Workshops, LNCS 9645, pp. 329–342, 2016.
DOI: 10.1007/978-3-319-32055-7_27

1. We present and compare an extended bloom filter [4] for a large dataset using MapReduce.
2. We consider the optimization of joins using mutual filtering policy based on extended Bloom Filter and conduct an extensive experimental evaluation.

The rest of this paper is organized as follows. We present related work in Sect. 2. In Sect. 3, we compute the Bloom Filter Using MapReduce. In Sect. 4, we present our approach for join operator. We will study how use the bloom filter to improve the efficiency of the join algorithm. In Sect. 5, experimental results demonstrating the performance of proposed join implementation are presented. We conclude the whole work in Sect. 6.

2 Related Work

In this section, we first review the Hadoop and join processing techniques in MapReduce. Then, we describe the Bloom filter.

2.1 Hadoop

Hadoop, the open source project of Apache, is a distributed parallel computing platform for large-scale data, including Hadoop Distributed File System (HDFS) and MapReduce.

MapReduce programs run on HDFS, which is the primary storage system used by Hadoop applications and provides high throughput access to application data. Data on HDFS is usually divided into many small blocks (splits). HDFS creates multiple replicas for each data block and distributes them on computing nodes throughout a cluster. One replica of a block would be processed by a Mapper locally on the node where it is distributed. This mechanism enables reliable and extremely rapid computations.

2.2 Joins in MapReduce

Join algorithms in MapReduce [12, 14] are roughly classified into two categories: map-side joins and reduce-side joins [5]. Map-side join algorithms are more efficient than reduce-side joins, because they only produce the final result of the join in map phase. However, they can be used only in particular circumstances. For Map-Merge join [6, 13], two input datasets should be partitioned and sorted on the join keys in advance, or an additional MapReduce job is required to meet the condition. Broadcast join [6] is effective when the size of one dataset is small.

Map-Reduce-Merge [7] adds merge phase after the reduce phase to support operations with multiple heterogeneous datasets, but it has the same drawback as reduce-side join algorithms. There are some attempts to optimize multiway joins in MapReduce. They discuss the same idea to minimize the size of the records replicated to reduce processes. In this paper, we address only two-way joins. However, our approach can be extended to multi-way joins by combining these work.

2.3 Bloom Filter

Bloom filter consists of an array of m bits and a set of k hash functions, which hash the element of the dataset to an integer in the range of [1, m]. The example for a bloom filter is shown in Fig. 1. All bits of the array are initialized to zero. Each hash function maps an element to some bits of the filter. In order to check the membership of an element, we must look at *k* positions. We answer positively only if all *k* bits are set to 1. The bloom filter allows false positives, but never false negatives.

Bloom join [4, 11] is a join algorithm which uses the Bloom filter to filter out tuples not matched in a join. Suppose relations *R(a, b)* and *S(a, c)* that reside in site 1 and site 2 respectively. In order to join these two relations, Bloom join algorithm generates a Bloom filter with the join key a of a relation, say *R*. Then, it sends the filter to site 2 where *R* resides. At the site 2, the algorithm scans *R* and sends the only tuples which are set in the received Bloom filter to site 1. Finally, a join of the filtered *R* and *S* is performed at site 1.

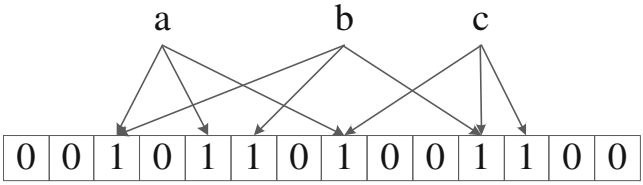

Fig. 1. Example for a bloom filter

Give a set *R(x)*, m is the size of the bit array, k is the number of hash functions, *p* is a false positive, we denote the bloom filter for the relation *R* on the attribute *x* by *BFR(x)*.

$$m_p = \frac{-\ln p}{(\ln 2)^2}$$

The total size of the bloom filter for the whole set *R(x)* is

$$m = m_p * |R| = \frac{-|R| * \ln p}{(\ln 2)^2}$$

3 Computing Bloom Filter Using MapReduce

A Bloom filter 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 bits in the array are initially set to 0. When an element is inserted into 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. To test membership of an element, if all bits

of its k hash positions of the array are 1, we can conclude that the element is in the set. Bloom filter may yield false positives, but false negatives are not generated.

The false positive of standard bloom filter:

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

Because of the false positive of standard bloom filter, many attributes that are not matched with the join condition have been transferred to the reduce phase. That will bring some of the network transmission overhead, *I/O* overhead. Therefore if we can further reduce the false positive of the bloom filter, so we can use bloom filter to decrease the network overhead between the map and reduce phases so as to improve the efficiency.

On the basis of the data structure, we define a new bit array of m. All bits in the array are initially set to 0. The element is hashed k times with k hash functions, we performed *XOR* operations on the results. The position in the array corresponding to the result is set to 1.

In the extend bloom filter, in this array, the probability of assigning a value to 1 is 1/m. When the hash address of each element of the XOR operation is mapped to the array, One bit is the probability of 0:

$$p' = \left(1 - \frac{1}{m}\right)^n \approx e^{-\frac{n}{m}}$$

So the false positive of the extend bloom filter:

$$p = \left(1 - e^{-\frac{kn}{m}}\right)^k * \left(1 - e^{-\frac{n}{m}}\right)$$

The map phase: Each map function build a bloom filter for its local data of its own partition named *BFR(x)*. $|BFR(x)| = m_p * |R|$. And the map function also make the XOR bloom filter named *BFK(x)*. The intermediate results of the map function output will be sent to a single reducer.

The reduce phase: The reduce function unions the intermediate results by a bit-wise OR operation.

The example of the strategy is shown in Fig. 2. There are two relation *R(A, B)* and *S (B, C)*. First, the two tables of all the local filter files for "OR" operation. Such as the relation R has m slices. We build $BFR(B_1)$ and $BFK(B_1)$, $BFR(B_2)$ and $BFK(B_2)$… $BFR(B_m)$ and $BFK(B_m)$ in the map phase. We unions the result by OR operation in the reduce phase. We build *BFR(B)* and *BFK(B)* for the relation *R(A,B)* and use *BFR(B)* and *BFK(B)* to filter the relation *S(B, C)*. We do the same to the relation *S*.

4 Extend Bloom Join Using MapReduce

In this section, we will study how to use the bloom filter to improve the efficiency of the join algorithm. The concept of using the bloom filter to improve the efficiency is based on the semi-join technique.

4.1 Two-Way Joins Using MapReduce

Aiming at the shortage for processing join operations based on MapReduce, we proposed the optimized strategy. When processing two tables join based on MapReduce model, we use mutual filtering policy based on extended bloom filter. After the extended bloom filter process, the two tables join attribute values are extracted respectively, and form the file. Then the files are used to filter two tables do not meet the join condition. The optimized method of join, can be achieved to extend bloom filter and reduce the false positive rate, reducing shuffle phase time, improving the execution efficiency of the system.

We present a algorithms using the extend bloom filter to compute *R(A, B)* and *S(B, C)*, each record of R and S has two attributes. This algorithm has two phases, each corresponding to a separate.

MapReduce job. In the first MapReduce job, we build a extend bloom filter *EBF(R)* on the attribute *B* for the relation *R* and *S*. The extend bloom filter *EBFR(R)* consists of *BFR(B)* and *BFK1(B)* for the relation R. The relation S includes *BFS(B)* and *BFK2(B)*. *BFK(B)* is the array which the hash address of each element of the *XOR* operation is mapped to. *BFK(B)* can reduce the false positive of the bloom filter.

In the second MapReduce job, Map function reads the two relations *R* and *S*, while reading *EBFR(B)* and *EBFS(B)*. For the elements in the relation *S*. The map function uses *EBFR(B)* to filter the relation S. The detection of the attribute is true means that the values of the join attributes need to send out. In the map phase, to express the natural join *R(A, B)* and *S(B, C)*, the input to the map function is key-value pairs *(r, t)*, where *r* is the relation name (either *R* or *S*), and t is a tuple of the relation named by *r*. the output is a key-value pair *(r, t)*. All these output records form the intermediate result.

The key is a composite of two elements. The first element is the value of the join attribute *B*, the second element is a flag bit, indicates that this property is derived from the relation *S*. The value contains an element, its value is the value of the attribute *B*. We do the same to the relation *R*. The map function uses *EBFS(B)* to filter the relation *R* for the elements in the relation *R*.

In the suffer phase, because the key is a composite value, If the entire key value is used as a partition element, It cannot guarantee that the same value is sent to the same reduce node and satisfy the equijoin condition. So we define a Partition function in advance, the first value of the key is used as the partition element. In the reduce phase, although the same key value is assigned to the same reduce, But extend bloom filter has the possibility of a miscarriage of justice too. So we need to add a step in the reduce function to check if the join property is equal. The final results are computed in the reduce function just like the improve repartition join described in [9].

The algorithm of Two-Way Joins Using MapReduce is as follows.

Algorithm 1 Two-Way Joins Using MapReduce

Input: the relation R *R(A, B)*,the relation S *S(B, C)*,the extend bloom filter *EBFR(B)*,the extend bloom filter *EBFS(B)*

Output: the result of joins r $R \bowtie S$

```
1   map(key = relationName, value = tuple)
2     joinAttrs = tuple.getValues(B);
3     if relationName == R then
4       if(EBFS(joinAttrs)==true) then
5         nonJoinAttrs = tuple.getValues(A);
6         output(joinAttrs+ relationName, nonJoinAttrs+relationName);
7       end if
8     end if
9     if relationName = S then
10      if(EBFR(joinAttrs)==true) then
11        nonJoinAttrs = tuple.getValues(C);
12        output(joinAttrs+ relationName, nonJoinAttrs+relationName);
13      end if
14    end if
15    partition(pair p, values)
16      hashcode = hashfunc(p.firstKey)
17      return hashcode % reducersNumber
18  reduce(key=joinAttrs+relationName, value=list(nonJoinAttrs+relationName))
19    rList = new List();
20    tag=key.getSecond();
21    for each (nonJoinAttrs+relationName) in value then
22      if(relationName == tag) then
23        rList.add(nonJoinAttrs);
24      end if
25      else then
26        for each a in rList then
27          output(a + key.getfirst() + nonJoinAttrs);
28        end for
29      end else
30    end for
```

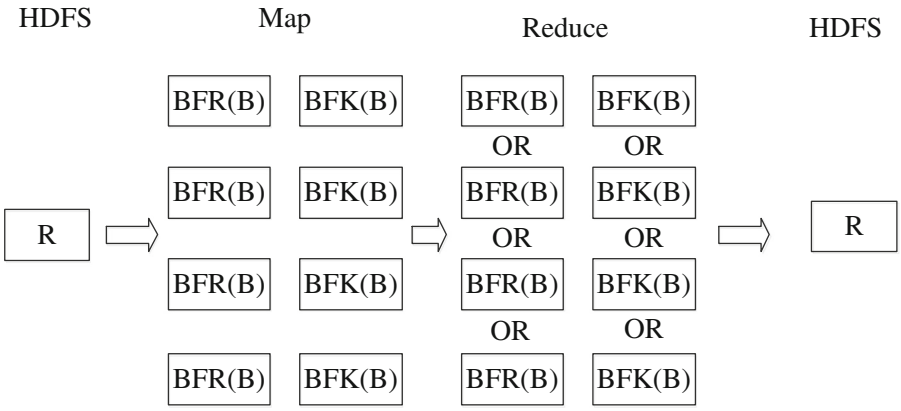

Fig. 2. Example for a extend bloom (*EBFR(B)*) filter generating

Mutual filtering strategy by improved Filter Bloom is mainly in order to filter out the two table does not meet the join conditions of the tuple, reduce the output of the map phase, greatly reduce the shuffle phase of the time, as well as the *I/O* overhead of the data. The efficiency of the system to perform the join task will be greatly improved.

Figure 3 is an example of the second phase of the algorithm. Relation *R* and *S* are stored in HDFS. The first phase of the MapReduce program for relation *R* generates *EBFR(B)*. and it do the same thing for the relation *S*. The second stage is shown in Fig. 3. Each map function reads the block of relation *R* and *S*. The map function reads the *EBFR(B)* and *EBFS(B)*. The map function uses it *EBFR(B)* to filter relation *S*. Therefore, the values of join attribute *B (6, 7, 8, 9)* are filtered out without passing through the network transmission in the relation *S*, just send a value of 1 and 2 to the reduce phase.

4.2 Multi-way Joins Using MapReduce

In this section, we solve the problem how to use bloom filters to a multi-way joins. Let us consider the case of a 3-way joins: $R(A, B) \bowtie S(B, C) \bowtie T(C, D)$. We can implement this join by a sequence of two two-way joins, choosing either to join *R* and *S* first, and then join T with the result. In [10], Afrati proposed another algorithm to deal with this join using only one MapReduce job. However, there are still a lot of tuples to be copied in this process. Naturally, we can use extend bloom filters to filter useless data to improve the efficiency of the multi-way joins.

We introduce an algorithm called multi-way-bf join having two MapReduce phases. In the first MapReduce phase, we build *EBFS(B)* and *EBFS(C)* for the relation *S* on the attribute *B* and *C*. In the second MapReduce phase, we use the *EBFS(B)* and *EBFS (C)* and adopt the algorithm [10] to get the final results.

The algorithm of the Replicated join with extend bloom filter as follows.

Algorithm 2 Multi-way Joins Using MapReduce

Input: the relation *R R(A, B),*the relation *S S(B, C),*the relation *T T(C, D),*the extend bloom filter *EBFS(B),* the extend bloom filter *EBFS(C)*

Output: the result of (key, value)

```
1   map(key = relationName, value = tuple)
2     joinAttrs = tuple.getValues(B);
3     if(relationName == R) then
4       if(EBFS(joinAttrs)==true) then
5         nonJoinAttrs = tuple.getValues(A);
6         output(joinAttrs+ relationName, value);
7       end if
8     end if
9     else if(relationName = T) then
10      if(EBFR(joinAttrs)==true) then
11        nonJoinAttrs = tuple.getValues(C);
12        output(joinAttrs+ relationName, value);
13      end if
14    end if
15    else then
16        nonJoinAttrs = tuple.getValues(D);
17      output(joinAttrs+ relationName, value);
18    end else
```

The algorithm of the Replicated join with extend bloom filter as follows.

Algorithm 3

Input: the (key, value), the number of partitions numPartitions

Output: the array of (key ,value)

```
1   Partition(pair p, value, int numPartitions)
2    We are connected to the attribute column B, C set a hash bucket value n1,n2
3       n1*n2=numPartition;
4       sList=new List();
5       if(p.secondKey ∈ R) then
6         H(B) = hash(value.B) % n1;
7         for i = 0 in n2 - 1 then
8           sList[i] = H(B) + i * n2;
9         end for
10        end if
11        else if(p.secondKey ∈ R) then
12          H(B) = hash(value.B) % n1;
13          H(C) = hash(value.C) % n2;
14          List[0] = H(B) + H(C) * n2;
15        end else if
16        else then
17          H(C) = hash(value.C) % n2;
18          for i = 0 in n1 – 1 then
19            sList[i] = H(C) + i * n1;
20          end for
21        end else
22        return List;
```

The algorithm of the Replicated join with extend bloom filter as follows.

Algorithm 4 The reduce of Multi-way Joins Using MapReduce

Input: (key ,value)

Output: the result of joins $R \bowtie S \bowtie T$

```
1   reduce(key=joinAttrs+relationName, values=list(value))
2     rList = new List();
3     sList = new List();
4     for each value in values then
5       if(value.tag == R) then
6         rList.add(value);
7       end if
8       else if(value.tag==S) then
9         sList.add(value);
10      end else if
11      else then
12      for each a in rList then
13        for each b in sList then
14          if(a.B==b.B&&b.C==value.C) then
15            output(a + b + value);
16          end if
17        end for
18      end for
19    end for
```

The general process of the algorithm is as follows, Let H be a hash function. The hash range is from 1 to n. Use *(i, j)* label each reduce, the range of values of I and j is 1 to n. Each element *S(B, C)* is sent to the label *(h(B), h(C))* for the reduce, Each element R(A, B) is sent to the label *(h(B), *)* for the reduce, * represents any value, Each element *T(C, D)* is sent to the label *(*, h(C))* for the reduce. In the end, the equivalent connection operation is done in each reduce.

5 Experiments

Our experiments run on a cluster consisting of 1 master node (served as both Namenode and Jobtracker) and 3 slave nodes. The release of Hadoop is 2.6.0. The size of each data block, which has 3 replicas on HDFS, is no more than 64 MB. In this

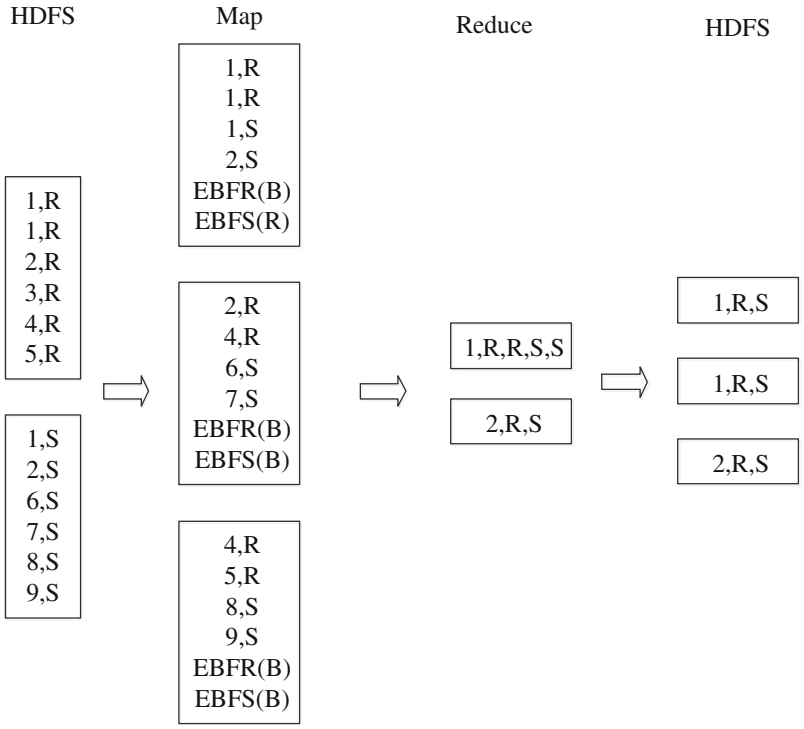

Fig. 3. Example of the second phase of the algorithm

paper, the data used in the experiments are derived from TPC-H. We compare our algorithm with the algorithm called EBF-M and EBF-RSJ.

We used the default Hadoop join RSJ and the EBF-RSJ to join these two relations on the cluster respectively. In the experiment of two table joins, we use relation ORDERS and CUSTOMER to do the join experiment in the TPC-H. Six sets of data of different sizes are selected in the experiment. The different datas are show as follows. Figure 4 shows the performance of the two joins (Table 1).

Table 1. Six groups of data for two relations

Data	1	2	3	4	5	6
CUSTOMER	160M	225M	525M	750M	900M	1.1G
ORDERS	175M	275M	400M	760M	940M	1.2G

From Fig. 4, when the size of the relation is small, RSJ is as the same as EBF-RSJ, because they add the additional MapReduce rounds to waste time. When the size of the relation is bigger, EBF-RSJ is more efficient than RSJ, because this filters a lot of useless data to save network overhead and processing overhead.

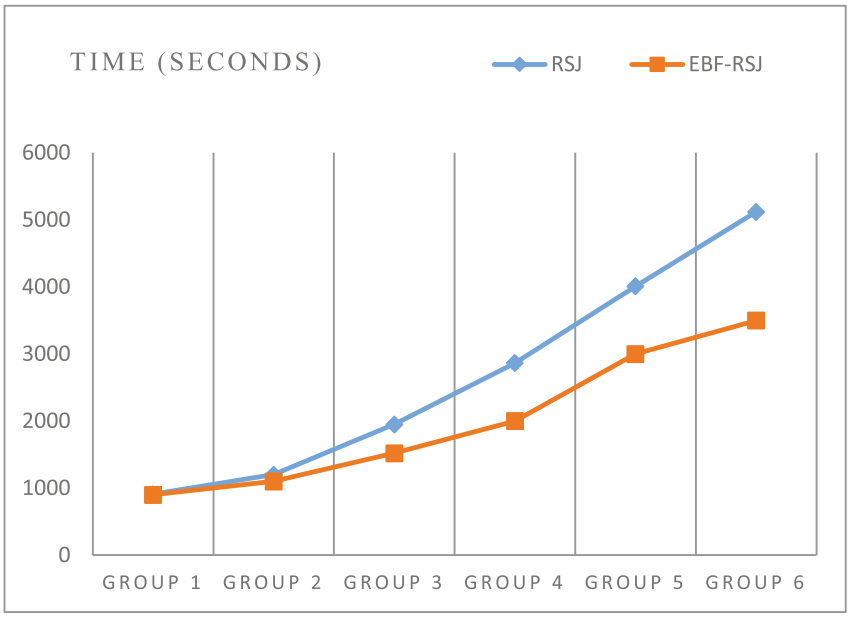

Fig. 4. MapReduce time for the default Hadoop join and extend Bloom Filter Join

For joins of three (or more) relations, the default Hadoop join has to join two relations and then join the intermediate result with the third relation. Figure 5 shows the performance comparison of Hadoop join and EBF-M. The different datas are show as follows (Table 2).

Table 2. Six groups of data for three relations

DATA	1	2	3	4	5	6
CUSTOMER	65M	125M	200M	400M	640M	780M
ORDERS	70M	100M	240M	390M	520M	680M
LINEITEM	81M	110M	234M	350M	510M	620M

We compare our algorithm with the algorithm called EBF-M and the default Hadoop join. The result is shown in Fig. 5. From Fig. 5, we can observe that our method is as efficient as the default Hadoop join when the relations are small, because our method adds an additional MapReduce phase to build the bloom filters. While the size of relations become large, our method is more efficient, as our method uses the extend bloom filter to filter a lot of useless data to save the network overhead and processing overhead.

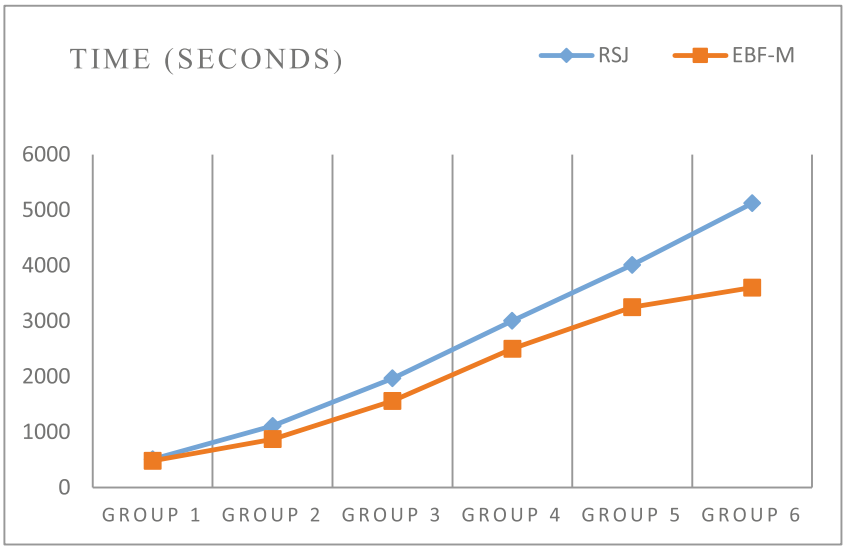

Fig. 5. MapReduce time for multi-way joins

6 Conclusion

In this paper, We present and compare an extended bloom filter for a large dataset using MapReduce, and we present a way to join relations within MapReduce framework by extend bloom filter. For joins of two relations, our approach avoids sorting large data records and reduces network overhead. It proves more efficient than the default join algorithm provided by Hadoop. For multi-joins, which the default join in Hadoop does not support directly, our method can improve the performance of multi-way joins.

In the future, we are planning to develop a dynamic cost analyzer. This will help us to implement a best MapReduce approach to any multi-way joins problems. We are also planning to investigate techniques of incorporating Hadoop parameters into the cost model to improve the join efficiency.

Acknowledgements. This paper was partially supported by National Sci-Tech Support Plan 2015BAH10F01 and NSFC grant U1509216, 61472099, 61133002.

References

1. Lueebber D, Grimmer U.: Systematic development of data mining based data quality tools. In: 29th VLDB (2003)
2. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: OSDI, pp. 137–150 (2004)
3. Apache Software Foundation. Hadoop, April 2010. http://hadoop.apache.org

4. Mackert, L.F., Lohman, G.M.: R* optimizer validation and performance evaluation for distributed queries. In: Proceedings of the 12th International Conference on Very Large Data Bases (VLDB), pp. 149–159 (1986)
5. Lee, K.-H., Lee, Y.-J., Choi, H., Chung, Y.D., Moon, B.: Parallel data processing with MapReduce: a survey. ACM SIGMOD Rec. **40**(4), 11–20 (2011)
6. Blanas, S., Patel, J.M., Ercegovac, V., Rao, J., Shekita, E.J., Tian, Y.: A comparison of join algorithms for log processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD 2010), pp. 975–986 (2010)
7. Yang, H.-C., Dasdan, A., Hsiao, R.-L., Parker, D.S.: Map-reduce-merge: simplified relational data processing on large clusters. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD 2007), pp. 1029–1040 (2007)
8. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM (CACM) **13**(7), 422–426 (1970)
9. 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: SIGMOD, pp. 975–986 (2010)
10. Afrati, F.N., Ullman, J.D.: Optimizing multiway joins in a map-reduce environment. IEEE Trans. Knowl. Data Eng. **23**(9), 1282–1297 (2011)
11. Broder, A., Mitzenmacher, M.: Network applications of bloom filters: a survey. In: Internet Mathematics, pp. 636–646 (2002)
12. Lee, K.-H., Lee, Y.-J., Choi, H., Chung, Y.D., Moon, B.: Parallel data processing with MapReduce: a survey. In: SIGMOD, pp. 11–20 (2011)
13. Yang, H.C., Dasdan, A., Hsiao, R.-L., Parker, D.S.: Map-reduce-merge: simplified relational data processing on large clusters. In: SIGMOD 2007, pp. 1029–1040 (2007)
14. Friedman, E., Pawlowski, P., Cieslewicz, J.: SQL/MapReduce: a practical approach to self-describing, polymorphic, and parallelizable user-defined functions. In: Proceedings of VLDB (2009)