Join Processing Using Bloom Filter in MapReduce

Taewhi Lee, Kisung Kim, and Hyoung-Joo Kim
School of Computer Science and Engineering, Seoul National University
1 Gwanak-ro, Seoul, Republic of Korea
{twlee, kskim}@idb.snu.ac.kr, hjk@snu.ac.kr

ABSTRACT

MapReduce is a programming model which is extensively used for large-scale data analysis. The join operation is one of the essential operations for the data analysis. However, MapReduce is not very efficient to perform the join operation since it always processes all records in the datasets even in the cases that only small fraction of datasets are relevant for the join operation. We alleviate this problem by applying bloomjoin algorithm, a classic distributed join algorithm. We improve the join performance using Bloom filters in MapReduce. In our approach, the Bloom filters are constructed in distributed fashion and are used to filter out redundant intermediate records. In order to apply the Bloom filters in MapReduce, we modify Hadoop to assign the input datasets to map tasks sequentially, and we propose a method to determine the processing order of input datasets based on the estimated cost. Our experimental results show that the number of intermediate results is decreased and the join performance can be improved in our architecture.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Distributed databases, Parallel databases, Query processing*

General Terms

Design, Performance

Keywords

Join processing, MapReduce, Hadoop, Bloom filter

1. INTRODUCTION

MapReduce [7] has been extensively used for large-scale data analysis in both academic and business areas. It can process huge amount of data in a reasonable time using a large number of commodity hardwares, and so valuable information hidden in those big data can be revealed with much less cost compared to previous techniques. The major benefits of MapReduce are a simple programming interface and extremely high scalability combined with graceful failure handling.

Unfortunately, MapReduce has some limitations to perform a join operation on multiple datasets, one of the essential operations for practical data analysis [5, 19]. The main problem of join processing in MapReduce is that the entire datasets should be processed and sent among nodes in the cluster via the network connection. This could cause a significant performance bottleneck, especially when only a small fraction of data is relevant for the join. In the database area, many techniques, such as semijoin [4] and bloomjoin [13], have been studied over the past 30 years [8] to address this problem. However, in MapReduce, any auxiliary data structures such as indexes or filters are not available because it is initially designed to process only a single large dataset as its input [7]. In this regards, some researchers criticized that MapReduce ignores rich technologies in database management systems, including efficient indexes and careful query execution planning [17].

In this work, we adopt bloomjoin algorithm [13] into the Hadoop [1], an open-source implementation of the MapReduce framework, to improve the join performance. We only consider join operations for two datasets. The idea is to construct Bloom filters [6] on one of the two input datasets and to filter out redundant records in the other dataset with the constructed filters in the map phase. By using this method, we can avoid processing redundant records and reduce communication overhead. However, it is not trivial to apply Bloom filters in MapReduce for following two reasons. First, the processing order of the input datasets in a MapReduce job cannot be controlled because MapReduce schedules the map tasks regardless of the dataset from which their corresponding input splits were read [19, 9]. Second, Bloom filters should be constructed in distributed fashion since an input dataset is divided into multiple splits and distributed to all nodes in the cluster.

In order to apply Bloom filters, we modify Hadoop as follows. First, we modify the map task scheduling policy so that input datasets for joins are processed sequentially in the map phase. In our approach, the processing cost is affected by the processing order of the input datasets. Therefore, we propose a method to choose the processing order based on the estimated cost. Second, we design and implement the execution flow to construct Bloom filters dynamically within a single MapReduce job. The execution flow consists of two phases: local filter construction and global filter merging.

RACS'12 October 23-26, 2012, San Antonio, TX, USA.
Copyright 2012 ACM 978-1-4503-1492-3/12/10 ...$15.00.

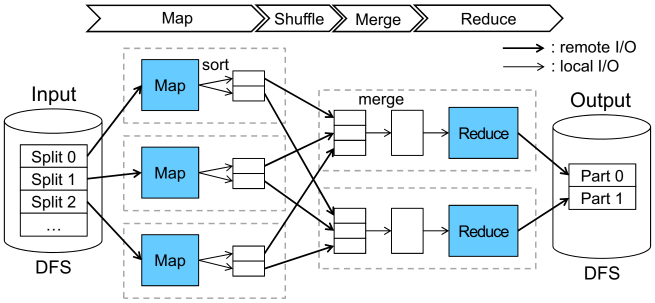


Figure 1: Execution overview of MapReduce.

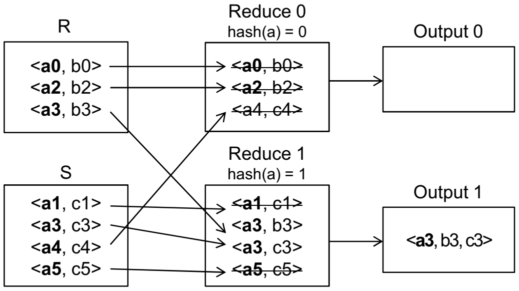


Figure 2: Basic join processing in MapReduce.

The rest of the paper is organized as follows: Section 2 introduces the background and related work to this paper. Section 3 describes our system architecture and implementation. In Section 4, we address the issues of cost estimation and fault tolerance. Then we present our experimental results in Section 5. Finally, we conclude and discuss future work in Section 6.

2. BACKGROUND AND RELATED WORK

In this section, we first review the MapReduce framework and join processing techniques in MapReduce. Then, we describe the Bloom filter and previous work improving join performance using the Bloom filter in different environments.

2.1 MapReduce

MapReduce [7] is a programming model for large-scale data processing run on a shared-nothing cluster. Since the MapReduce framework provides automatic parallel execution on a large cluster of commodity machines, users only need to specify their program logic without thought of parallel and distributed processing.

A MapReduce program consists of two functions: `map` and `reduce`. The `map` function takes a set of records from input files as simple key/value pairs and produces a set of intermediate key/value pairs. The values in these intermediate pairs are automatically grouped by key and passed to the `reduce` function. For this grouping, sort and merge processes are conducted. The `reduce` function takes an intermediate key and a set of values corresponding to the key, and then it produces final output key/value pairs. An execution overview of MapReduce is shown in Figure 1.

A MapReduce cluster is composed of one master node and a number of worker nodes. The master periodically communicates with every worker using a heartbeat protocol to check their status and control their actions. When a MapReduce job is submitted, the master splits the input data and creates map tasks for the input splits, and reduce tasks. The master assigns each task to idle workers. A map worker reads the input split and executes `map` function specified by the user. A reduce worker should read the intermediate pairs from all map workers and execute `reduce` function. When all tasks are complete, the MapReduce job is finished.

2.2 Joins in MapReduce

Join algorithms in MapReduce are roughly classified into two categories: map-side joins and reduce-side joins [12]. 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 [12], 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 [5] is effective when the size of one dataset is small.

Reduce-side join algorithms is more general, but they are not efficient because the whole input records have to be sent from map workers to reduce workers. Figure 2 shows an example of a join between R(a,b) and S(a,c) with a basic reduce-side join algorithm, which is called repartition join in [5]. All of input records are collected by reduce workers to join the records with the same key. During this process, redundant records (marked with strikethrough in Figure 2) are also collected. Semijoin described in [5] requires three MapReduce jobs in general, so it has to process input datasets multiple times.

Map-Reduce-Merge [19] 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 multi-way joins in MapReduce [3, 9]. 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

A Bloom filter [6] 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 merit of Bloom filter is space efficiency. The size of the Bloom filter is fixed regardless of the number of the elements n, but there is a tradeoff between m and the false positive probability p. The probability of a false positive after inserting n elements can be calculated as follows [6]:

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

Bloomjoin [13] 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, Bloomjoin 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. We adopt this algorithm to the MapReduce framework.

The bloomjoin algorithm is combined with a group-by operation and extended to multi-way joins in [10]. More recent studies [14, 18] optimizes complex distributed multi-way joins using this algorithm. However, while all these work assumes the join relations are not partitioned, in MapReduce environment, a dataset are split and distributed in many nodes. Therefore, it is not trivial to construct Bloom filters in MapReduce environment. Although reduce-side join with Bloom filter is proposed in [16], it requires three MapReduce jobs like semijoin in [5]. The work described in [11] is closely connected with our paper. This work conducts the theoretical investigation to apply the bloomjoin algorithm within a single MapReduce job, but it does not provide concrete technical details. We implement a working system and present performance evaluation results of our approach.

3. PROPOSED ARCHITECTURE

This section describes the overall architecture of our implementation and the major changes we have made. We have implemented our approach into Hadoop [1], an open-source implementation of the MapReduce framework. In Hadoop, the master node is called jobtracker and the worker node is called tasktracker. We will use the terms of Hadoop in the remaining of this paper.

3.1 Execution Overview

Figure 3 shows the overall execution flow of a join operation on the datasets R and S in our implementation. In this example, we suppose that R is chosen to be processed first; that is, Bloom filters are built on R. We use the term “build input” for an input dataset processed first and “probe input” for the other dataset. When the user program is submitted, the following sequence of actions is performed.

1. **Job submission.** If a job is submitted, m_1 map tasks for R, m_2 map tasks for S, and r reduce tasks are created. A task includes all necessary information to be run on a tasktracker such as the job configuration and the location of the corresponding input/output files.

2. **First map phase.** The jobtracker assigns the m_1 map tasks or the reduce tasks to idle tasktrackers. A map tasktracker reads the input split for the task, converts it to key/value pairs, and then executes the `map` function for the input pairs.

3. **Local filter construction.** The intermediate pairs produced from the `map` function are divided into r partitions, which are sent to r tasktrackers respectively. Additionally, r Bloom filters are constructed on the keys in each partition. These filters are called *local filters* because they are built for only the intermediate results in a single tasktracker. If a tasktracker runs multiple map tasks, it merges the local filters of each task and maintains only r filters.

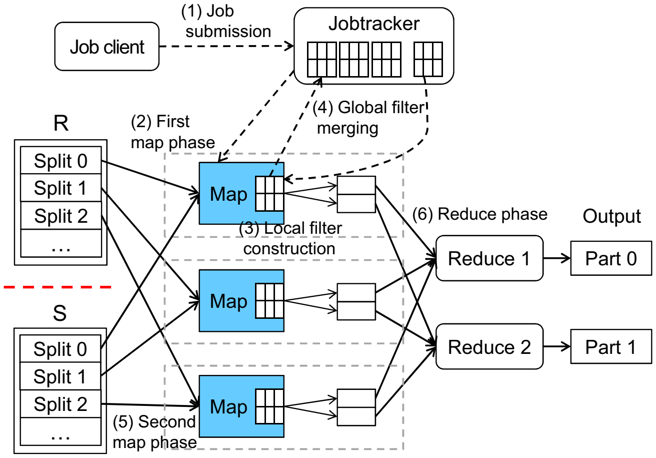


Figure 3: Execution overview.

4. **Global filter merging.** When all m_1 map tasks are complete, the jobtracker signals all tasktrackers to send the local filters via heartbeat responses. Then, all tasktrackers send their local filters to the jobtracker, and the jobtracker constructs the *global filters* for the dataset R. Next, the jobtracker sends the global filters to all tasktrackers. Until building and tranmitting the global filters are complete, the map tasks for the dataset S are not assigned.

5. **Second map phase.** The jobtracker assigns the m_2 map tasks or the remaining reduce tasks to the tasktrackers. Tasktrackers run the assigned task with the received global filters. The input key/value pairs which are not set in the global filters are filtered out.

6. **Reduce phase.** This step is the same as the reduce phase in Hadoop. A reduce tasktracker reads the corresponding intermediate pairs from all map tasktrackers using remote procedure calls. It sorts the all intermediate pairs and runs the `reduce` function. Final output results are written in the given output path.

We have made two modifications to the design of Hadoop. First, we schedule map tasks in the order of the dataset. Second, we construct Bloom filters on the build input in distributed fashion to filter out the probe input. The following subsections describe more details on these points.

3.2 Map Task Scheduling

Hadoop basically assigns map tasks in the order of the input split size, considering the locality of the input split. Consequently, map tasks on different input datasets are intermingled by the task scheduler. In our approach, map tasks are assigned according to a certain order of input datasets. That is, after processing one dataset is finished, the other dataset is began to be assigned for map tasks. In one dataset, map tasks are assigned as in the original Hadoop.

In order to schedule map tasks by the datasets, each map task needs to contain the dataset ID of the corresponding input split, so we implement a `DataSetInputFormat` and a `DataSetSplit` class by extending a `FileInputFormat` and a

`FileSplit` classes, respectively. In addition, we modify the `JobQueueTaskScheduler` class to schedule map tasks with the dataset ID. The task scheduler assigns the m_1 map tasks only for the build input among all map tasks. If no more map task for the build input is left, the task scheduler defers the assignment of the m_2 map tasks for the probe input until the jobtracker constructs the global filters and broadcasts them to all tasktrackers.

This policy gives us the opportunity to apply database techniques such as tuple filtering and join ordering. However, in this policy, all tasktrackers cannot run the map tasks for the probe input until the global filter construction is finished (of course, they can run the copy operations of reduce tasks or the tasks of other MapReduce jobs). The waiting time could be long if there exist straggler nodes. In this case, Hadoop runs multiple copies of the same task on different tasktrackers when the job is close to completion. This feature is called the speculative execution. We can reduce the waiting time using the speculative execution during the global filter construction phase.

3.3 Bloom Filter Construction

While a tasktracker runs a map task of the build input, it creates the Bloom filters on the intermediate records produced from the task. A Bloom filter is created for each map output partition assigned for each reduce task, and therefore the total number of the Bloom filters is the number of reduce tasks. We use the `MurmurHash` implemented in Hadoop as the hash function and set the number of hash function k is 2, which can be changed by the users. When multiple map tasks are run on a tasktracker, the tasktracker merges each Bloom filters from the tasks and maintains only one set of Bloom filters. We call this set of Bloom filters as local filters.

When all map tasks for the build input are complete, The jobtracker has to gather all local filters to construct the global filters. We add two `TaskTrackerAction` classes, called `SendLocalFilterAction` and `ReceiveGlobalFilterAction`, for the global filter construction. The jobtracker sends the `SendLocalFilterAction` as the heartbeat response to all tasktrackers, and they send the jobtracker their local filters. The jobtracker merges all the local filters to build the global filters using bitwise OR operations, and sends the `ReceiveGlobalFilterAction` with the global filters in the heartbeat response to all tasktrackers.

The communication cost for the global filter construction C_f is

$$C_f = 2c_t \cdot m \cdot r \cdot t \tag{2}$$

where c_t is the cost to transfer data from one node to another, m is the size of a Bloom filter, r is the number of reduce tasks, and t is the number of tasktrackers. The coefficient 2 is multiplied since the local filters and the global filters are transmitted between the jobtracker and the tasktrackers.

If r or t is large, C_f can be large, then the global filter construction could be a bottleneck. We can distribute the overhead for merging the local filters by merging them hierarchically, which is not implemented yet. We leave this issue for future work.

4. COST ANALYSIS

We adjust the cost model for Hadoop described in [15] to consider the cost for the construction of Bloom filters. We use the same assumption of the cost model that the execution time is dominated by I/O operations. The query optimizer can use this cost model to determine the processing order of input datasets and whether to use Bloom filters.

4.1 A Cost Model

Assume that we have a MapReduce job that read the input dataset R and S. Let $|R|$ be the size of R, $|S|$ be the size of S, and $|D|$ the size of the intermediate data of the 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, and the size of the sort buffer is $B + 1$.

The total cost for a join operation between R and S is the sum of the cost C_s to perform the sorting and merging at the map and reduce nodes, the cost C_f to transfer Bloom filters among the nodes as shown in Equation 2, and the cost C_t to transfer intermediate data among the nodes. We omit the cost to read the input and to write the final output because they are the same regardless of using Bloom filters. Then, the total cost C is as follows:

$$C = C_s + C_f + C_t \tag{3}$$

where

$$C_s = c_l|D| \cdot 2(\lceil \log_B |D| - \log_B (m_1 + m_2)\rceil + \lceil \log_B (m_1 + m_2)\rceil)$$

$$C_t = c_t \cdot |D|$$

as in [15].

We add the cost C_f to the cost model described in [15]. C_f is constant since the Hadoop parameters and the size of a Bloom filter are set. If Bloom filters are not used, C_f is zero. Another difference in our model is the size of intermediate results $|D|$ is changed according to the performance of Bloom filters, and this is discussed in the following section.

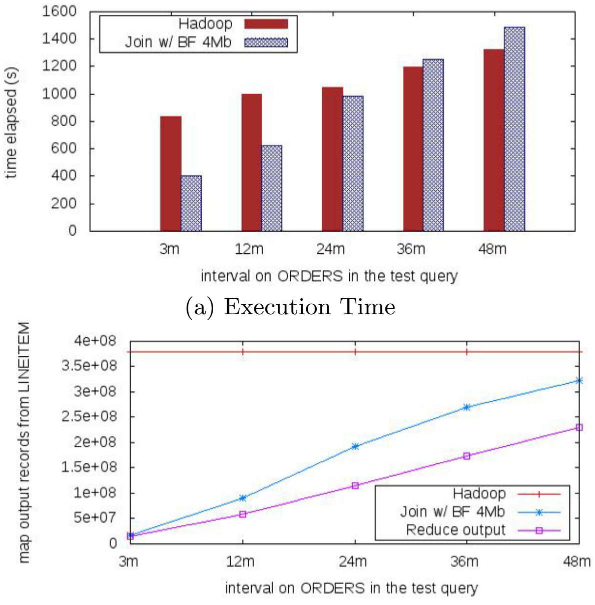
4.2 Determining the Processing Order

The processing order of input datasets is important in our approach, because the total cost of a join depends on the processing order. We define the best processing order as the order which minimizes the total cost.

The important factor that determines the total cost is the size of intermediate results $|D|$. In order to estimate $|D|$, we need to know the number of distinct join keys and the ratio of joined records in each dataset, and we assume that the information is already given (e.g. using metadata for input datasets). Let σ_R is the ratio of the joined records of S with R, and σ_S is the ratio of the joined records of R with S. Then we can estimate $|D|$ with the probability of a false positive p from Equation 1 as follows:

$$|D| = \begin{cases} |R| + \sigma_R|S| + p(1 - \sigma_R)|S| & \text{if } \mathsf{R} \text{ is the build input} \\ |S| + \sigma_S|R| + p(1 - \sigma_S)|R| & \text{if } \mathsf{S} \text{ is the build input} \\ |R| + |S| & \text{if Bloom filter is not used} \end{cases}$$

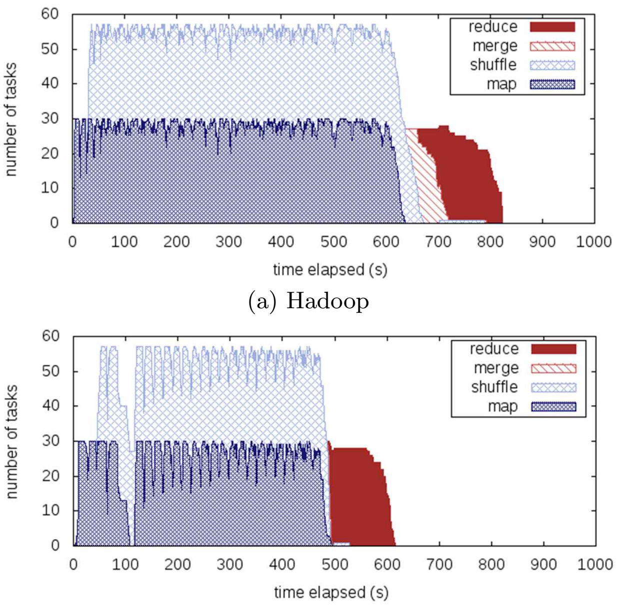
We compute the total cost for above three cases with Equation 3 and choose the best order among them. However, the total cost does not include the global filter merging time because we could not expect the time which map tasks take to be complete and it cannot be measured consistently. Accordingly, we approximate the cost as the time that all tasktrackers execute one map task though this is a very simple approximation. We compare the total cost for the three cases after adding the approximate cost to the cost for the two cases using Bloom filters.



(a) Execution Time

(b) The number of intermediate results

Figure 4: Join Performance.



(a) Hadoop

(b) Join with Bloom Filter 4Mb

Figure 5: Task timelines.

5. EXPERIMENTAL RESULTS

In this section, we present experimental results of our implementation. All experiments were run on a cluster of 11 machines that consists of 1 jobtracker and 10 tasktrackers. Each machine has 3.1GHz quad-core CPU, 4GB memory, and 2TB hard disk. The operating system is 32-bit Ubuntu 10.10, and the java version we used is 1.6.0_26.

We implement the proposed architecture on Hadoop 0.20.2. We set the HDFS block size to 128MB and the replication factor to 3. Each tasktracker can simultaneously run 3 map tasks and 3 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 TPC-H benchmark [2] 100GB dataset to evaluate our implementation against the original Hadoop. We join the two tables of `lineitem` and `orders` which have 600M records and 150M records respectively. The `orderkey` column of `lineitem` table is a foreign key to `orderkey` column of `orders` table. Also, we add some selection predicates in brought from TPC-H Q4 query to control the join selectivities. The following join query is performed on the dataset.

```
SELECT *
FROM lineitem l, orders o
WHERE l.orderkey = o.orderkey
  AND l.commitdate < l.receiptdate
  AND o.orderdate >= 1992-01-01
  AND o.orderdate < 1992-01-01 + interval '?' months
```

Table 1: The number of intermediate/output records

interval months	map output from |orders|	map output from |lineitem|	reduce output |orders⋈lineitem|
3	5.7M	379M	14.3M
12	22.8M		57.7M
24	45.6M		115.2M
36	68.3M		172.8M
48	91.1M		230.3M

We run the query, changing the interval '?' months to 3, 12, 24, 36, and 48 months. Hadoop programs for the join queries are hand-coded, and the summary of the datasets for each query is given in Table 1. We choose `orders` as the build input and `lineitem` as the probe input based on the estimated cost.

5.2 Evaluation

We perform each test query on the proposed architecture with a Bloom filter size of 4Mb. Figure 4(a) shows the execution time of the test queries using Hadoop and our implementation. We can observe that our implementation shows better performance than Hadoop when the number of records in the build input and joined records is small. The more records in the build input, the less redundant records we can filter out. As the interval on orders table is large, the number of intermediate results is increased as shown in Figure 4(b). In addition, we observe the number of false positives, the gap between 'Join w/ BF 4Mb' line and 'Reduce output', is gradually increased.

Figure 5(a) and 5(b) shows the task timelines of Hadoop and our implementation during the execution of the test query with the interval of 12 months. A key observation is that the number of running map tasks is sharply decreased for a while in Figure 5(b). It means that tasktrackers do not run the map tasks for the second input dataset during the global filter construction phase. In spite of the overhead of this period, the execution time of all map tasks and reduce tasks is considerably reduced. Map phase is finished early because the intermediate results are reduced by Bloom filters, and as a result, the local I/O and sorting cost are also reduced. Reduce phase is also finished early because the number of intermediate records to be copied from remote map processes is reduced, so the number of input records to process in `reduce` function is decreased.

Next, we perform the test query with the interval of 12 months, changing the size of a Bloom filter from 512Kb to 4Mb. Figure 6 shows the execution time for the experiments.

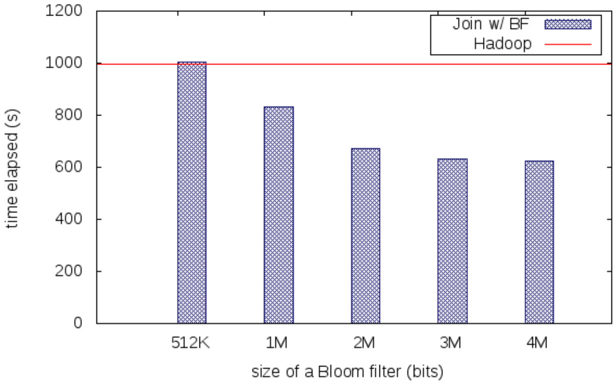


Figure 6: Performance in different Bloom filter size.

If the size of Bloom filters is too small, redundant records cannot be filtered out. On the other hand, if the size of the filters is too large, the overhead to construct and communicate the filters may be large. It is important to determine the appropriate size of the Bloom filter, and we leave this for future work.

6. CONCLUSION AND FUTURE WORK

In this paper, we have presented an architecture to improve the join performance using Bloom filters in the MapReduce framework. We have made two design changes to Hadoop. First, we assign map tasks in the order of the dataset. Second, we construct Bloom filters in distributed fashion. We have evaluated our architecture by running the join queries to TPC-H dataset on our commodity cluster. The results show that our architecture significantly improves the execution time of join queries, especially in the case that join selectivity and the number of distinct join keys are small.

In future work, we will extend our architecture to support multi-way joins, and implement the optimizer module to decide the size of Bloom filters and the processing order of input datasets automatically. Also, we plan to apply some other filtering techniques to our architecture.

7. ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MEST)(No. 20110017480).

8. REFERENCES

[1] http://hadoop.apache.org/.

[2] http://www.tpc.org/tpch/.

[3] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology (EDBT '10)*, pages 99–110, 2010.

[4] P. A. Bernstein and D.-M. W. Chiu. Using semi-joins to solve relational queries. *Journal of the ACM (JACM)*, 28(1):25–40, 1981.

[5] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*, pages 975–986, 2010.

[6] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM (CACM)*, 13(7):422–426, 1970.

[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th USENIX Symposium on Opearting Systems Design & Implementation (OSDI)*, pages 137–150, 2004.

[8] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)*, 25(2):73–169, 1993.

[9] D. Jiang, A. K. H. Tung, and G. Chen. Map-join-reduce: Toward scalable and efficient data analysis on large clusters. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 23(9):1299–1311, 2011.

[10] A. Kemper, D. Kossmann, and C. Wiesner. Generalized hash teams for join and group-by. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*, pages 30–41, 1999.

[11] P. Koutris. Bloom filters in distributed query execution. http://www.cs.washington.edu/education/courses/cse544/11wi/projects/koutris.pdf, 2011.

[12] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel data processing with mapreduce: A survey. *ACM SIGMOD Record*, 40(4):11–20, 2011.

[13] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB)*, pages 149–159, 1986.

[14] L. Michael, W. Nejdl, O. Papapetrou, and W. Siberski. Improving distributed join efficiency with extended bloom filter operations. In *Proceedings of the 21st International Conference on Advanced Networking and Applications (AINA)*, pages 187–194, 2007.

[15] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. Mrshare: Sharing across multiple queries in mapreduce. *Proceedings of the VLDB Endowment*, 3(1-2):494–505, 2010.

[16] K. Palla. A comparative analysis of join algorithms using the hadoop map/reduce framework. Master's thesis, University of Edinburgh, 2009.

[17] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. Dewitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD '09)*, pages 165–178, 2009.

[18] S. Ramesh, O. Papapetrou, and W. Siberski. Optimizing distributed joins with bloom filters. In *Proceedings of the 5th International Conference on Distributed Computing and Internet Technology (ICDCIT)*, pages 145–156, 2008.

[19] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: Simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD '07)*, pages 1029–1040, 2007.