Exploiting Bloom Filters for Efficient Joins in MapReduce *

Taewhi Lee, Kisung Kim, and Hyoung-Joo Kim
School of Computer Science and Engineering, Seoul National University
1 Gwanak-ro, Seoul 151-742, Republic of Korea
{twlee, kskim}@idb.snu.ac.kr, hjk@snu.ac.kr

Abstract

MapReduce is a programming model that is extensively used for large-scale data analysis. However, it is inefficient to perform join operations using MapReduce, because large intermediate results are produced, even in cases where only a small fraction of input data participate in the join. We alleviate this problem by exploiting Bloom filters within a single MapReduce job. We create Bloom filters for an input dataset, and filter out the redundant records in the other input dataset in the map phase. To do this, we modify the MapReduce framework in two ways. First, map tasks are scheduled according to the processing order of input datasets. Second, Bloom filters are dynamically created in a distributed fashion. We propose two map task scheduling policies and provide a method to determine the processing order based on the estimated cost. Our experimental results show that the proposed techniques decrease the size of intermediate results and can improve the execution time.
Keywords : join processing, MapReduce, Hadoop, Bloom filter

1 Introduction

MapReduce [8] has been extensively used for large-scale data analysis in both academic and business areas. It can process huge amounts of data in a reasonable time using a large number of commodity hardwares, and so valuable information hidden in this data can be revealed with a lower cost compared to previous techniques, such as traditional databases. The major benefits of MapReduce are its simple programming interface and extremely high scalability combined with graceful failure handling.

Unfortunately, MapReduce has some limitations to performing a join operation on multiple datasets, one of the essential operations for practical data analysis [6, 23]. The primary problem of join processing in MapReduce is that it emits large intermediate results, regardless of the number of final join results. This could cause an unnecessary network overhead for sending the intermediate results to other cluster nodes, and a disk I/O cost for sorting and merging them, even when only a small fraction of input data participate in the join.

*A preliminary version of this paper entitled "Join Processing Using Bloom Filter in MapReduce" has been presented at the 2012 ACM Research in Applied Computation Symposium (RACS'12). This paper is an extended version that includes an additional map task scheduling policy and new experimental results.

In the field of databases, many techniques have been studied over the past 30 years to address this problem [9]. However, in MapReduce, auxiliary data structures such as indexes or filters are not available, because it was initially designed to process a single, large dataset [8]. In this regard, some researchers have criticized MapReduce for ignoring rich technologies in database management systems, including efficient indexes and careful query execution planning [19]. To apply such techniques, the modified MapReduce versions [6, 18] of semijoin [5] and bloomjoin [15] have been proposed, but they require multiple MapReduce jobs to process input data multiple times.

In this work, we propose join techniques that utilize Bloom filters within a single MapReduce job. We consider join operations for two datasets in this paper. Our fundamental idea is to create Bloom filters [7] on one input dataset, and to filter out redundant records in the other dataset by applying these filters in the map phase. In this way, we can reduce the communication cost for redundant records by processing the input datasets only once. However, it is not trivial to apply Bloom filters within a single MapReduce job for the following two reasons. First, the processing order of input datasets cannot be controlled in the original MapReduce framework, because MapReduce schedules map tasks regardless of the dataset from which their corresponding input splits were obtained. Second, the Bloom filters should be constructed in a distributed fashion, because an input dataset is divided into multiple splits and distributed to all cluster nodes.

To resolve these issues, we modify the MapReduce framework as follows. First, we change the map task scheduling so that input datasets are processed sequentially in the map phase. Second, we design the execution flow to construct Bloom filters dynamically within a single MapReduce job. Locally created Bloom filters for an input dataset are sent to, and merged at, a master node. We have implemented these features on the top of Hadoop [1], an open-source implementation of the MapReduce framework. On this framework, we define two map task scheduling policies: synchronous and asynchronous scheduling. In synchronous scheduling, map tasks for the second input dataset are not assigned during the merging phase. In contrast, under asynchronous scheduling, the map tasks are continuously assigned, though some tasks can be processed without the merged filters. Under our scheduling policies, the processing cost is affected by the processing order of the two input datasets. Therefore, we propose a method to choose the processing order based on the estimated cost.

The rest of this paper is organized as follows. Section 2 introduces the background and related work to this study, and Section 3 describes the design and implementation of our proposed framework. In Section 4, we provide a method to determine the processing order of input datasets based on the estimated cost. Then, we present our experimental results in Section 5. Finally, we conclude and discuss future work in Section 6.

2 Background and Related Work

We first review the MapReduce framework and its join processing techniques. Then, we describe the Bloom filter and previous work aimed at improving join performance using the Bloom filter in different environments.

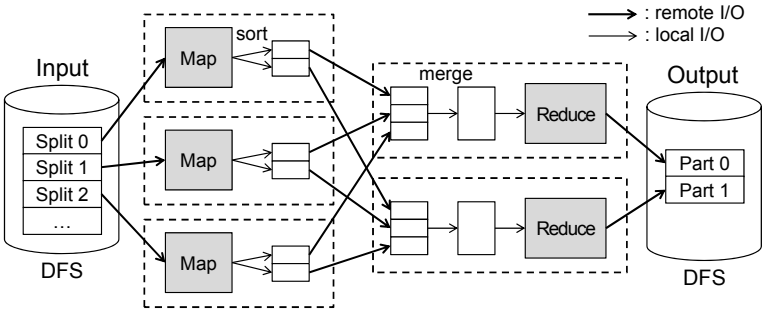


Figure 1: Execution overview of MapReduce.

2.1 MapReduce

MapReduce [8] is a programming model for large-scale data processing run on a shared-nothing cluster. As the MapReduce framework provides automatic parallel execution on a large cluster of commodity machines, users can easily write their programs without the burden of implementing features for 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. Sort and merge operations are involved in this grouping process. The `reduce` function takes an intermediate key and a set of values corresponding to the key, and then 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 the workers using a heartbeat protocol to check their status and control their actions. When a MapReduce job is submitted, the master creates map and reduce tasks, and then assigns each task to idle workers. A map worker reads the input split and executes the `map` function specified by the user. A reduce worker reads the intermediate pairs from all map workers and executes the `reduce` function. When all tasks are complete, the MapReduce job is finished.

2.2 Bloom Filter

A Bloom filter [7] 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 the membership status of an element, we then evaluate this array. If all bits of the element's k hash positions are 1, we can conclude that it is in the set. Bloom filters may yield false positives, but false negatives are not produced.

The merit of the Bloom filter is its space efficiency. Its size is fixed regardless of the number of 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 [7]:

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

The Bloom filter has been used for efficient join processing. Bloomjoin [15] is a join algorithm to filter out tuples not matched by a join using Bloom filters. Suppose that relations R(a,b) and S(a,c) reside in site 1 and site 2, respectively. In order to join these two relations, the bloomjoin algorithm generates a Bloom filter with the join keys of a relation, say R. Then, it sends the filter to site 2, where S resides. At site 2, the algorithm scans S and sends only the tuples with the join keys that are set in the received filter to site 1. Finally, it joins R and the filtered S at site 1. We adopt this algorithm for the MapReduce framework.

Bloomjoin was combined with a group-by operation and extended to multi-way joins in [12]. More recent studies [16, 20] optimize complex distributed multi-way joins using this algorithm. However, these techniques assume that the join relations are neither split nor distributed to other nodes dynamically, which is not the case in the MapReduce environment.

2.3 Joins in MapReduce

Join algorithms in MapReduce are roughly classified into two categories: map-side joins and reduce-side joins [14]. Map-side joins are more efficient than reduce-side joins, because they confine the final join results to the map phase. However, they can only be used in particular circumstances. For Hadoop's map-side join [22], called the Map-Merge join [14], 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. A broadcast join [6] is efficient only when the size of one dataset is small.

Reduce-side joins can be generally used, but they are inefficient because large intermediate 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 the basic reduce-side join algorithm, called the repartition join [6]. In this example, all of the input records are collected by reduce workers to join records with the

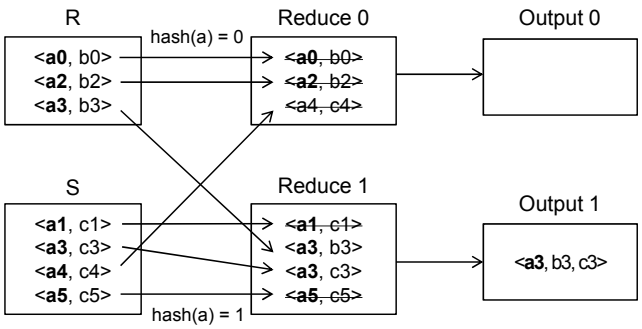


Figure 2: Basic join processing in MapReduce.

same key, including redundant records marked with strikethrough text. Semi-join [6] requires three MapReduce jobs. It finds unique join keys from an input dataset, finds joined records in the other dataset, and then produces final join results with an independent MapReduce job. Reduce-side joins with a Bloom filter are described in [18], but these create the Bloom filter via an independent job. Therefore, it is necessary to process input datasets multiple times. Our proposed technique is completed within a single MapReduce job. [13] is closely connected with our paper. This work conducts the theoretical investigation of using Bloom filters within one MapReduce job, but does not provide concrete technical details. We implement a working system, and propose synchronous and asynchronous scheduling.

Map-Reduce-Merge [23] adds a merge phase after the reduce phase to support operations with multiple heterogeneous datasets, but suffers from the same drawback as reduce-side join algorithms. There have been some attempts to optimize multi-way joins in MapReduce [4, 11]. They discuss the same idea of minimizing the size of the records replicated to the reduce workers. In this paper, we address only two-way joins. However, our approach can be extended to multi-way joins by combining these approaches.

3 The Proposed Framework

This section describes the overall architecture of our framework 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 the *jobtracker* and the worker node is called the *tasktracker*. We will use these terms in the remainder of this paper.

3.1 Execution Overview

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

1. **Job submission.** If a job is submitted, map and reduce tasks are created. Assume that 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 enable it 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 for R 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. For each partition, a Bloom filter is created by inserting the keys of its intermediate pairs. These filters are called *local filters*, because they are

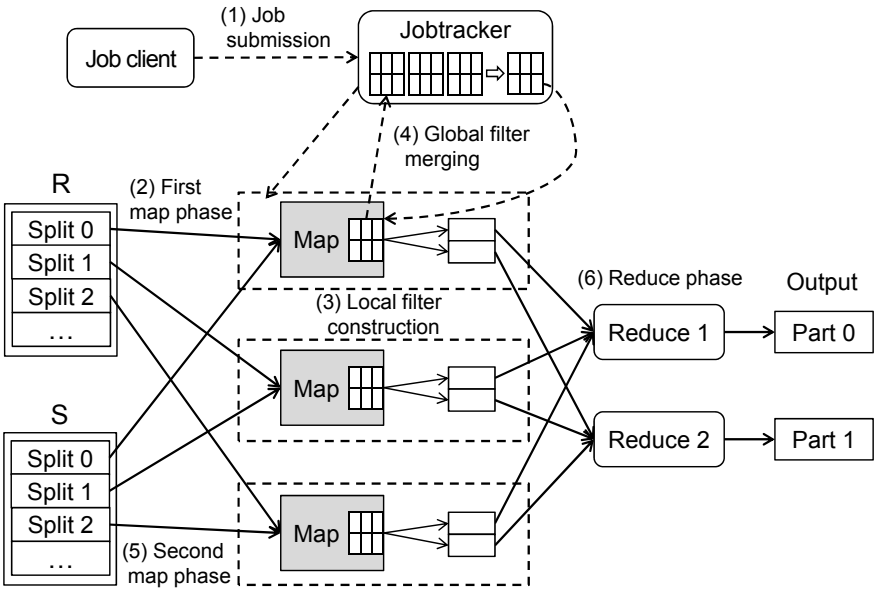


Figure 3: Execution overview.

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.

4. **Global filter merging.** When all m_1 map tasks are complete, the jobtracker signals all tasktrackers to send it their local filters via heartbeat responses. Then, the jobtracker merges all local filters to construct the *global filters* for R. Next, the jobtracker sends the global filters to all tasktrackers.

5. **Second map phase.** The jobtracker assigns the m_2 map tasks for S or the remaining reduce tasks to the tasktrackers. Tasktrackers run the assigned tasks with the received global filters. The intermediate pairs with keys that are not present 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 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 according to the order of the dataset. Second, we construct Bloom filters on the build input in a 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 based on 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. We assign map tasks according to a certain order of the input datasets. This gives us the opportunity to apply database techniques such as tuple filtering and join ordering. The order is determined based on the estimated cost, as described in Section 4. Within a single input dataset, map tasks are assigned as in the original Hadoop.

In order to schedule map tasks by dataset, each map task needs to know the dataset identifier of the corresponding input split. Accordingly, we have implemented new input format and input split classes (`DataSetInputFormat` and `DataSetSplit`) to contain this information by extending the respective Hadoop `FileInputFormat` and `FileSplit` classes. In addition, we have modified `JobQueueTaskScheduler`, which is the default task scheduler of Hadoop, to schedule map tasks using the dataset identifier.

We define the two scheduling policies of synchronous and asynchronous scheduling. These are similar in that they assign map tasks in the order of the input datasets, but their behavior is different during the global filter merging phase.

3.2.1 Synchronous Scheduling

Under the synchronous scheduling policy, our task scheduler does not assign the map tasks for the probe input during the global filter merging phase. Instead, the assignment is deferred until the global filters are constructed and sent to the tasktrackers. Then, every probe input split can be processed with the filters, so more redundant intermediate results can be filtered out.

However, under this policy, all tasktrackers cannot run the map tasks for the probe input, and should wait 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, especially if straggler nodes exist. Then, the gain from the filtering is offset by the loss from such waiting. Hadoop has a feature known as speculative execution, in which multiple copies of the same task are run on different tasktrackers when the job is close to completion. We can reduce the waiting time using speculative execution during the global filter merging phase.

3.2.2 Asynchronous Scheduling

Even if the waiting time during the global filter merging phase is not long, the loss from the waiting may be large depending on the size of the records that are filtered out. Thus, we provide an asynchronous scheduling policy. Under asynchronous scheduling, our task scheduler continues to assign the map tasks for the probe input without such waiting. Tasktrackers can run these tasks without filtering until they receive the global filters.

In this policy, tasktrackers do not need to wait during the global filter merging phase. Instead, the size of the intermediate results may be increased. There is a tradeoff between synchronous and asynchronous scheduling, and it depends on the waiting time and the filter performance.

3.3 Bloom Filter Construction

While a tasktracker runs a map task for 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, which is assigned for each reduce task. Therefore, the total number of Bloom filters is the number of reduce tasks. When multiple map tasks are run, each tasktracker merges its Bloom filters, so that only one set of Bloom filters is maintained. We call this set the local filters.

When all map tasks for the build input are complete, the jobtracker must gather all local filters to construct the global filters. In Hadoop, the jobtracker controls the tasktrackers by putting some instructions, called `TaskTrackerAction`, in heartbeat messages. For the merging process, we add two `TaskTrackerAction` classes, called `SendLocalFilterAction` and `ReceiveGlobalFilterAction`. 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.

If the number of reduce tasks or tasktrackers is large, the global filter construction can become a bottleneck. This overhead could be distributed by merging local filters hierarchically, although this has not yet been implemented—we leave this issue for future work.

3.4 API and Parameters

Hadoop provides a library class called `MultipleInputs` to support MapReduce jobs that have multiple input paths with a different `InputFormat` and `Mapper` for each path. This class is convenient, as it allows users to specify which jobs should perform the join operation. We provide a similar library class called `JoinInputs`. Users can specify jobs to be joined with Bloom filters using the following API.

```
JoinInputs.addInputPath(conf, path, dsid, inputformat, mapper)
```

Compared to `MultipleInputs`, we add `dsid` to specify the dataset identifier, and use `inputformat` as a subclass of `DataSetInputFormat`, described in 3.2.

We also add several parameters to configure our framework, as shown in Table 1. Users can define these in Hadoop configuration files, or specify them as runtime parameters.

Table 1: User Parameters

Parameter	Description	Type	Default value
mapred.filter.use	whether to use Bloom filters for join processing	boolean	true
mapred.filter.size	size of a Bloom filter (bits)	integer	4194304
mapred.filter.num.hash	number of hash functions for Bloom filters	integer	2
mapred.filter.async	asynchronous/synchronous map task scheduling policy	boolean	true

4 Cost Analysis

We adjust the cost model for Hadoop described in [17] to consider the cost for the construction of Bloom filters. We use the same assumption of the cost model, i.e., that the execution time is dominated by I/O operations. Using this cost model, the query optimizer can choose the strategy that minimizes the total cost, including the processing order of input datasets, the map task scheduling policy, and whether to use Bloom filters.

4.1 Cost Model

Assume that we have a MapReduce job for a join on R and S. Let $|R|$ be the size of R, $|S|$ be the size of S, and $|D|$ be the size of the intermediate data of the job. In addition, let c_r be the cost of reading or writing data remotely, c_l be the cost of reading or writing data locally, and c_t be the cost of transferring data from one node to another, as defined in [17]. We will assume that the size of the sort buffer is $B + 1$, the maximum size of the input split is b, and the maximum number of map tasks that are run simultaneously on the cluster is n_m.

The total cost for the job is the sum of the cost C_r of reading the input data, the cost C_s of performing the sorting and merging at the map and reduce nodes, the cost C_f of transferring the Bloom filters among the nodes, and the cost C_t of transferring intermediate data among the nodes. We omit the cost of writing the final results, because this is the same regardless of whether Bloom filters are used.

Then, the total cost C is as follows:

$$C = C_r + C_s + C_f + C_t \tag{2}$$

where

$$C_r = \begin{cases} c_r \cdot (|R| + |S| + b \cdot n_m) & \text{, if synchronous scheduling is used} \\ c_r \cdot (|R| + |S|) & \text{, otherwise} \end{cases} \tag{3}$$

$$C_s = c_l \cdot |D| \cdot 2(\lceil \log_B |D| - \log_B (m_1 + m_2) \rceil + \lceil \log_B (m_1 + m_2) \rceil) \tag{4}$$

$$C_f = 2c_t \cdot m \cdot r \cdot t \tag{5}$$

$$C_t = c_t \cdot |D| \tag{6}$$

Note that we compute C_r by adding $b \cdot n_m$ to the input data if synchronous scheduling is used in Equation 3. This cost model only considers I/O costs, but we should include the waiting time during the global filter merging phase in the cost model. Though it cannot be measured consistently, we approximate its overhead as the cost that the maximum number of map tasks can be run simulatenously on all tasktrackers. Accordingly, we include the cost by converting the waiting time into the maximum size of the input splits for the maximum number of map tasks.

C_s and C_t are the same as in [17], and we add the cost C_f to the cost model. C_f is a constant because the Hadoop 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 as local and global filters are transmitted between the jobtracker and the tasktrackers. Another difference in

our model is that the size of the intermediate results $|D|$ changes according to the scheduling policy and the performance of the Bloom filters. This is discussed in the following subsection.

4.2 Estimating the Size of the Intermediate Results

The most important factor in determining the total cost is the size of the 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. We assume that this information is given. It may be maintained in some systems. (In Hive [2], for example, some research has been conducted that aims to optimize queries using table and column statistics [21, 10].) Otherwise, this information can be determined from certain parameters.

For simplicity, we shall assume that the job does not have any selection predicate on the input datasets. Selection predicates affect the size of the intermediate results, but our model can be easily extended to consider them.

The size of the intermediate results depends on the map task scheduling policy described in 3.2. Let σ_R be the ratio of the joined records of S with R, and σ_S be that of R with S. In synchronous scheduling, $|D|$ can be estimated using the probability of a false positive p from Equation 1 as follows:

$$|D| = \begin{cases} |R| + \sigma_R|S| + p(1-\sigma_R)|S| & \text{, if } \mathsf{R} \text{ is the build input} \\ |S| + \sigma_S|R| + p(1-\sigma_S)|R| & \text{, if } \mathsf{S} \text{ is the build input} \end{cases} \tag{7}$$

In asynchronous scheduling, $|D|$ can be estimated as follows:

$$|D| = \begin{cases} |R| + |S|_a + \sigma_R \cdot |S|_f + p(1-\sigma_R) \cdot |S|_f & \text{, if } \mathsf{R} \text{ is the build input} \\ |S| + |R|_a + \sigma_S \cdot |R|_f + p(1-\sigma_S) \cdot |R|_f & \text{, if } \mathsf{S} \text{ is the build input} \end{cases} \tag{8}$$

where

$$|R|_a = min(b \cdot n_m, |R|), \quad |R|_f = |R| - |R|_a,$$
$$|S|_a = min(b \cdot n_m, |S|), \quad |S|_f = |S| - |S|_a.$$

In asynchronous scheduling, $|R|_a$ or $|S|_a$ give the size of the probe input splits that are processed without filtering, and $|R|_f$ or $|S|_f$ give the size of the probe input splits that are processed with Bloom filters. As some probe input splits are processed without filtering, the size of the intermediate results in asynchronous scheduling may be larger than that under synchronous scheduling. However, synchronous scheduling suffers from the waiting time during the global filter merging phase, and this cost is included in Equation 3.

Our cost model can be used to compute the total cost of the repartition join (without Bloom filters) with the condition that $C_f = 0$ and $|D| = |R| + |S|$. We can use our cost model to choose the processing order of the input datasets, the map task scheduling policy, and whether to use Bloom filters.

5 Experimental Results

In this section, we present experimental results of our implementation. All experiments were run on a cluster of 11 machines consisting of 1 jobtracker and

10 tasktrackers. Each machine had a 3.1 GHz quad-core CPU, 4 GB memory, and 2 TB hard disk. The operating system was 32-bit Ubuntu 10.10, and the Java version used was 1.6.0_26.

We implemented the proposed framework on Hadoop 0.20.2. We set the HDFS block size to 128 MB and the replication factor to 3. Each tasktracker could simultaneously run three map tasks and three reduce tasks. The I/O buffer was set to 128 KB, and the memory for sorting data was set to 200 MB.

5.1 Datasets

We used TPC-H benchmark [3] datasets, varying the scale factor (SF) between 100, 200, and 300. The scale factor describes the entire database size of the dataset in gigabytes. We performed a join between the two largest tables in the database, `lineitem` and `orders`. The `orderkey` column of the `lineitem` table is a foreign key to the `orderkey` column of the `orders` table. Thus, we added some selection predicates to control the join selectivities. Our test query can be expressed in SQL-like syntax as follows:

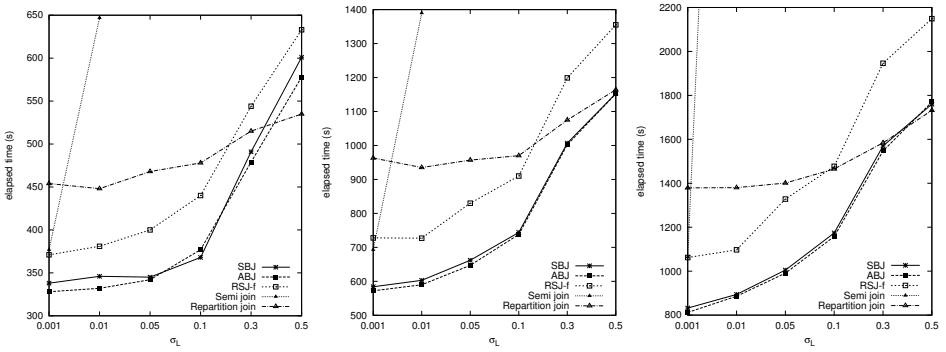
```
SELECT substr(l.*, 0, len), substr(o.*, 0, len)
FROM lineitem l, orders o
WHERE l.orderkey = o.orderkey
  AND o.custkey < '?'
```

We ran the query, changing the '?' in the predicate to set the ratio of joined records of `orders` with `lineitem` (σ_L) to between 0.001 and 0.5. In addition, we set the query results as substrings of the joined records in each table in order to vary the size of the intermediate results by changing the length of the substrings `len`. Hadoop programs for the test queries were hand-coded, and we chose `orders` as the build input and `lineitem` as the probe input based on the estimated cost.

5.2 Evaluation

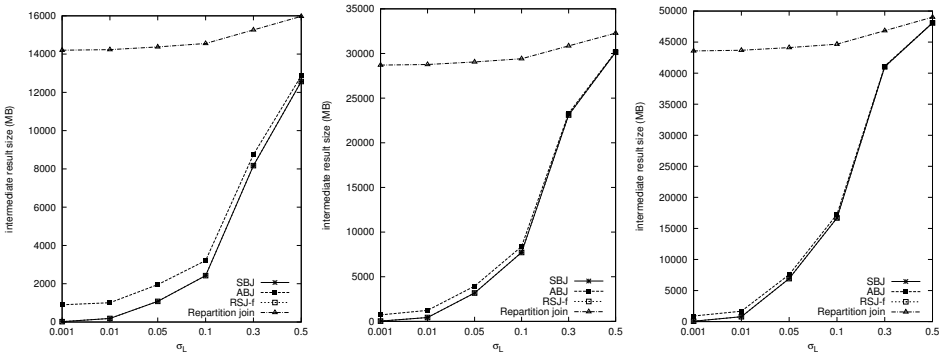
We compared the performance of our techniques to that of the existing repartition join [6], semijoin [6], and the reduce-side join with Bloom filters, which are created in an independent MapReduce job, referred to as RSJ-f [18]. Our techniques can be run with synchronous and asynchronous scheduling, and we refer to these as SBJ (Synchronous Bloom Join) and ABJ (Asynchronous Bloom Join). We used the `MurmurHash` implemented in Hadoop as the hash function, and set the number of hash function $k = 2$ and the size of a Bloom filter to 4 Mb. We also used this configuration for RSJ-f, so that the performance of the Bloom filters did not differ between tests.

Figure 4 shows the execution times of the test queries using each join technique on various sizes of TPC-H dataset. Figures 4(a), 4(b), and 4(c) show similar patterns. We can observe that the techniques using filters (SBJ, ABJ, and RSJ-f) show better performance than the repartition join when σ_L is small. Among these, our techniques outperform RSJ-f, which processes the build input twice and has additional costs to initialize and cleanup an extra MapReduce job. If more records participate in the join, the performance becomes worse because the number of redundant records that can be filtered out is reduced. Semijoin



(a) SF=100 (b) SF=200 (c) SF=300

Figure 4: Execution times on various sizes of TPC-H dataset. (`len` = 10, 128 MB HDFS block)



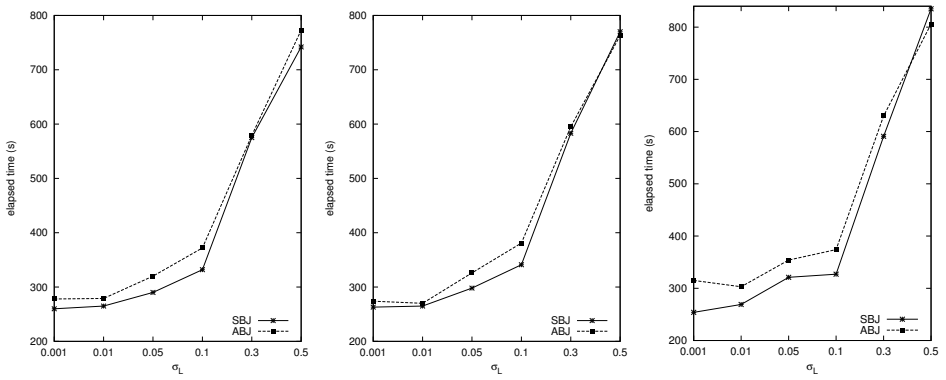
(a) SF=100 (b) SF=200 (c) SF=300

Figure 5: Intermediate result sizes on various sizes of TPC-H dataset. (`len` = 10, 128 MB HDFS block)

did not finish when σ_L was greater than or equal to 0.05, because it ran out of memory.

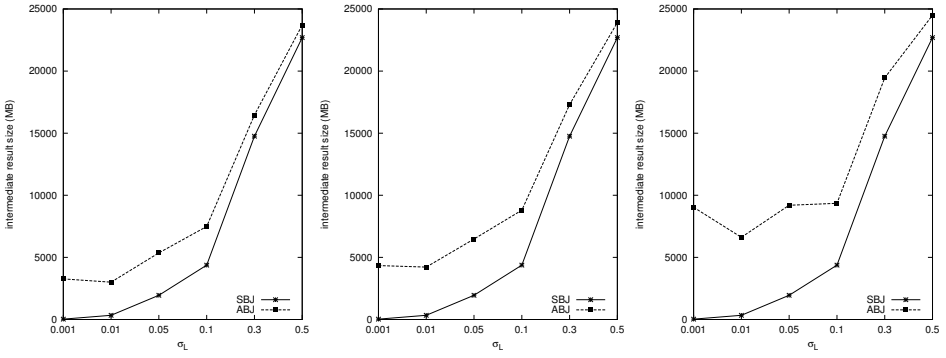
Figures 5(a), 5(b), and 5(c) show the intermediate result sizes in each test case. Those of ABJ vary each time, so we present the average size in our results. We exclude the semijoin because it uses a map-side join for the second and third jobs. In each case, SBJ and RSJ-f have almost the same size intermediate results, and ABJ has a slightly larger size because some map tasks are processed without filtering in the global filter merging phase. Instead, SBJ has the additional cost of waiting during the merging phase, and RSJ-f runs an extra MapReduce job to create the Bloom filters. The repartition join has the largest size, because it emits all probe input records as intermediate results. We can see that the intermediate result sizes increase as σ_L increases, and this leads to an increase in execution times. As the size of the Bloom filters is fixed, the probability of false positives increases with the number of join keys that are inserted into the filters. This is more obvious in the case of a large scale factor.

With the configurations in Figure 4 and 5, ABJ shows better performance than SBJ, although this is dependent on the size of the intermediate results that



(a) 256MB (b) 384MB (c) 512MB

Figure 6: Execution times with various HDFS block sizes. (SF = 100, `len` = 30)



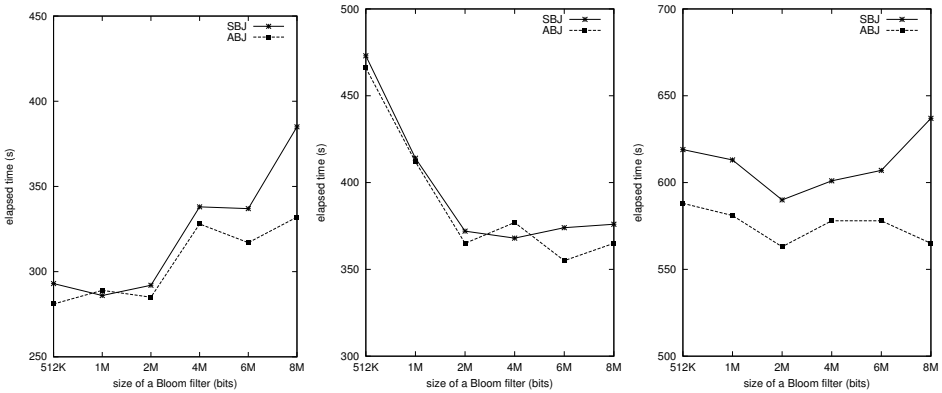
(a) 256MB (b) 384MB (c) 512MB

Figure 7: Intermediate result sizes with various HDFS block sizes. (SF = 100, `len` = 30)

are produced in the global filter merging phase. Figure 6 shows the execution times with various HDFS block sizes. As the block size is the maximum size of an input split in the map phase, ABJ processes more input records without filtering as the block size increases. We also set `len` to 30 to clarify the impact of the increase in intermediate result size. In this case, SBJ is better than ABJ when σ_L is small.

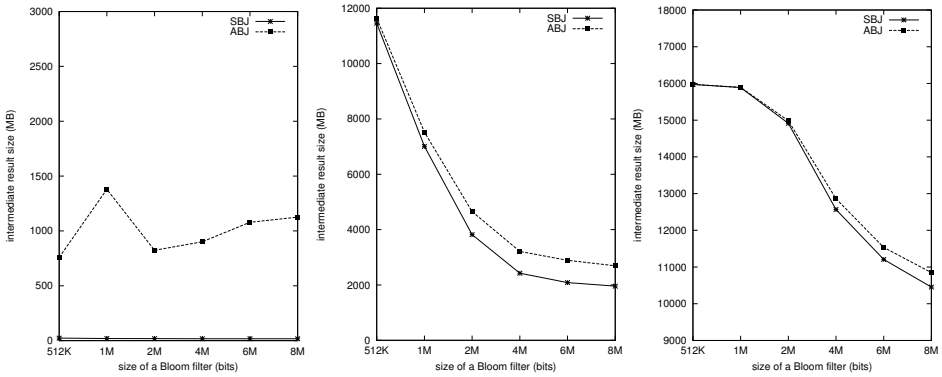
Figures 7(a), 7(b), and 7(c) show the intermediate result sizes corresponding to the cases in Figures 6(a), 6(b), and 6(c). We can observe that the difference between the intermediate result sizes of SBJ and ABJ is generally large when σ_L is small, and the gap gets larger as the block size gets larger. Therefore, we can confirm that the performance of SBJ and ABJ is affected by the intermediate result size, and is dependent on several factors such as the performance of the Bloom filters, the input split size, and the maximum number of map tasks, as described in Section 4.2.

Finally, we ran the test query with various sizes of the Bloom filter, from 512 Kb to 8 Mb. Figure 8 and Figure 9 show the execution times and intermediate



(a) σ_L=0.001 (b) σ_L=0.1 (c) σ_L=0.5

Figure 8: Execution times with various Bloom filter sizes. (SF = 100, `len` = 10, 128 MB HDFS block)



(a) σ_L=0.001 (b) σ_L=0.1 (c) σ_L=0.5

Figure 9: Intermediate result sizes with various Bloom filter sizes. (SF = 100, `len` = 10, 128 MB HDFS block)

result sizes, respectively, for the experiments. In Figure 9(a), we can observe that the 512 Kb Bloom filter was sufficient when σ_L=0.001, so the intermediate result sizes were barely reduced as the size of the Bloom filter increased. Rather, a large-sized filter increased the execution time, as shown in Figure 8(a). If the filter size is too large, the overhead for constructing and communicating the filters offsets the benefits of filtering. On the other hand, in Figure 9(c), we can observe that the 512 Kb Bloom filter was too small when σ_L=0.5, so only a small number of redundant records were filtered out. This also increases the execution time, as shown in Figure 8(c). If the size of the Bloom filters is too small, redundant records cannot be filtered out. Therefore, it is important to determine the most appropriate size for the Bloom filter. This can be determined from statistical information about the input datasets, and we leave this for future work.

6 Conclusions and Future Work

In this paper, we have presented an architecture to improve the join performance of the MapReduce framework using Bloom filters. We made two design changes to Hadoop. First, we assigned map tasks based on the order of the dataset. Second, we constructed Bloom filters in a distributed fashion. We have proposed two scheduling policies, synchronous and asynchronous scheduling, and described a cost model to estimate the total join cost in each case. We have evaluated our techniques against several existing join algorithms with various sizes of TPC-H dataset on our commodity cluster. The results show that our techniques significantly improve the query execution time, especially in the case where a small fraction of an input dataset is being joined.

In future work, we will extend our framework to support multi-way joins, and implement an optimizer module to automatically determine the best join strategy in terms of the processing order of the input datasets, the scheduling policy, and the Bloom filter size. Further, we plan to apply some other filtering techniques to our framework.

Acknowledgements

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 20120005695).

References

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

[2] Hive. `http://hive.apache.org/`.

[3] TPC-H benchmark. `http://www.tpc.org/tpch/`.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

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

[10] A. Gruenheid, E. Omiecinski, and L. Mark. Query optimization using column statistics in hive. In *Proceedings of the 15th Symposium on International Database Engineering & Applications (IDEAS '11)*, pages 97–105, 2011.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *Proceedings of the 26th IEEE International Conference on Data Engineering (ICDE '10)*, pages 996–1005, 2010.

[22] T. White. *Hadoop: The Definitive Guide*, pages 247–249. O'Reilly Media, Inc., second edition, 2011.

[23] 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.

Taewhi Lee received the B.S. degree in computer science and engineering from Seoul National University, Seoul, Korea, in 2004. He is currently a Ph.D. candidate in School of Computer Science and Engineering at Seoul National University, Seoul, Korea. His research interests include large-scale data processing, graph databases, and social data analysis.

Kisung Kim received the B.S. degree in chemical engineering from Seoul National University, Seoul, Korea, in 2003. He is currently a Ph.D. candidate in School of Computer Science and Engineering at Seoul National University, Seoul, Korea. His current research focuses on RDF data processing, graph databases, and parallel and distributed data processing.

Hyoung-Joo Kim received the B.S. degree in computer engineering from Seoul National University, Seoul, Korea, in 1982, and he received the M.S. and Ph.D. degrees in computer science from the University of Texas at Austin, USA, in 1985 and 1988, respectively. He was an assistant professor of the College of Computing at the Georgia Institute of Technology. He is currently a professor in School of Computer Science and Engineering at Seoul National University, Seoul, Korea. His research interests include large-scale data processing, social web, and ontology.