J Supercomput (2014) 69:793–813
DOI 10.1007/s11227-014-1179-9

Join processing with threshold-based filtering in MapReduce

Taewhi Lee · Hye-Chan Bae · Hyoung-Joo Kim

Published online: 9 April 2014

Abstract Data analytics, in particular those involving heterogeneous data, often require join operations on datasets collected from different sources. MapReduce, one of the most popular frameworks for large-scale data processing, is not suited for joining multiple datasets. This is because MapReduce often produces a large number of redundant intermediate results, irrespective of the size of the joined records. Although several existing approaches attempt to reduce the number of such redundant results using Bloom filters, they may be inefficient if large portions of records are joined or the number of distinct keys is large. To alleviate this problem, we propose a join processing method with threshold-based filtering in MapReduce, called TMFR-Join, which is an abbreviation for "Threshold-based Map-Filter-Reduce Join". TMFR-Join applies filters according to their performance, which is estimated in terms of false-positive rates. It also provides a general framework for exploiting various filtering techniques that support certain desired operations. The experimental results indicate that the performance of TMFR-Join is close to that of the better of existing join processing techniques, both with and without filters.

Keywords Join processing · Threshold-based filtering · MapReduce · Hadoop

T. Lee (✉)
BigData Software Platform Research Department, Electronics and Telecommunications Research Institute, 218 Gajeong-ro, Yuseong-gu, Daejeon 305-700, Republic of Korea
e-mail: taewhi@etri.re.kr

H.-C. Bae
Media Solution Center, Samsung Electronics Co., Ltd.,
129 Samsung-ro, Yeongtong-gu, Suwon-si, Gyeonggi-do 443-742, Republic of Korea
e-mail: hyechan.bae@samsung.com

H.-J. Kim
Department of Computer Science and Engineering, Seoul National University,
1 Gwanak-ro, Gwanak-gu, Seoul 151-744, Republic of Korea
e-mail: hjk@snu.ac.kr

1 Introduction

Large-scale data analysis plays an increasingly important role in business decision-making activities. Facebook collects tens to hundreds of terabytes of user log data every day, which it analyzes to provide a number of features, such as Facebook Insights for advertisers and friend recommendations [1]. Telecom companies generate billions of voice call data records per day, which are analyzed to enhance service quality [2]. Under certain circumstances, it is necessary to analyze heterogeneous datasets, which are collected from different sources. For example, Samsung Electronics produces a variety of devices, such as smartphones, tablet PCs, and televisions. Each device, or an application installed on it, accumulates different types of log data. These data have to be analyzed together to discover business insights. Consequently, the joining of large heterogeneous datasets has become an important issue.

MapReduce [3] is a very useful framework for large-scale data analysis. It facilitates the processing of tremendous amounts of data in a reasonable amount of time using a large cluster of commodity machines. It does this by providing a simple programming interface composed of map and reduce functions so that users can easily implement their jobs. For further convenience, MapReduce supports automatic parallel and distributed processing of user programs with robust failure-handling mechanisms. It is now more widely used with the emergence of Hadoop [4], an open-source implementation of the MapReduce framework.

MapReduce performs well on a single homogeneous dataset, but not for join operations on multiple heterogeneous datasets [5,6]. To join multiple datasets in MapReduce, all input records have to be sent from map workers to reduce workers, regardless of the size of the joined records, as shown in Fig. 1. This can produce a large number of redundant intermediate results that incur disk I/O costs for sort and merge and network I/O costs for communication with other cluster nodes [7].

A few researchers have tried to reduce the size of the redundant intermediate results using Bloom filters [8] for join processing in MapReduce [9–12]. Their approaches involve filtering out the intermediate results that are not joined, and perform efficiently

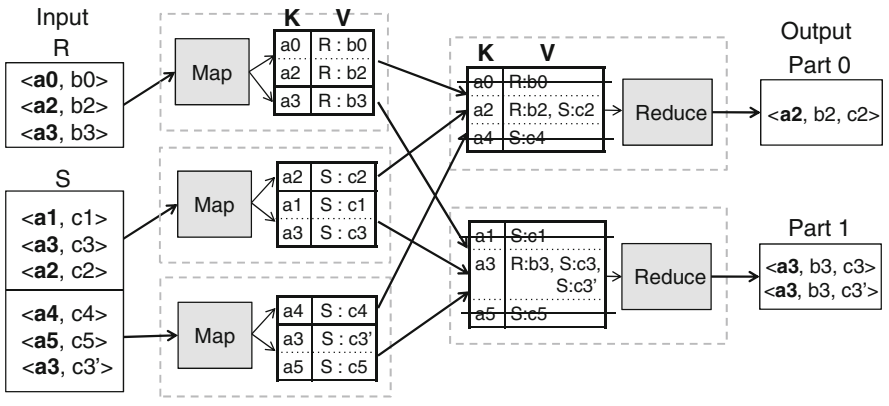

Fig. 1 Basic join processing in MapReduce. *Each dotted line box* denotes a map/reduce worker

if a small portion of input records is joined. However, they may perform worse than join processing without filters if large portions of records are joined or the number of distinct keys is large. Furthermore, if statistical information for input datasets is not available or inaccurate, they may be inefficient because filter parameters like the number of bits and the number of hash functions cannot be optimally adjusted. In such cases, applying the filters increases overhead because of the computing of hash values for each record and the merging of the filters.

To handle such cases, this paper proposes a join processing method with threshold-based filtering, called TMFR-Join. TMFR-Join monitors the performance of filters by means of false-positive rates (FPRs), and disables filters whose FPRs are greater than a user-configured threshold. It also facilitates the application of various other filtering techniques in addition to the Bloom filter. Any filters that support certain desired properties can be used in TMFR-Join. Note that we focus on applying filters based on a threshold for stable performance, not choosing the most efficient one that will vary with the input datasets and join queries. The contributions made in this paper are as follows:

- We propose TMFR-Join, a join processing method with threshold-based filtering, which applies filters according to their FPRs. The FPRs can be computed when it merges the filters that are created in all nodes. Because it is beneficial to detect poor filter performance as early as possible, it estimates the FPRs of merged filters with those of individual filters before they are merged.
- We extend the proposed approach to apply various filtering techniques that satisfy certain desired properties. We clarify the properties and show example filters including some Bloom filter variants [13], the Quotient filter [14], and the Interval filter [15].
- We define the best FPR threshold value as an equilibrium FPR threshold, and address how to compute it by analyzing the costs of join processing with and without filters based on a cost model.
- We implement a working TMFR-Join prototype in Hadoop [4] and evaluate our approach against existing join processing with and without filters. The experimental results show that our approach has performance that is close to the performance of the better of them.

The remainder of this paper is organized as follows: Sect. 2 reviews the background information and work related to this paper. Section 3 explains the design and implementation details of TMFR-Join. Section 4 addresses the computation of the best FPR threshold value based on a cost model. Section 5 discusses our experimental results. Finally, Sect. 6 concludes this paper.

2 Background and related work

2.1 Join processing in MapReduce

Join algorithms in MapReduce are roughly classified into two categories: map-side joins and reduce-side joins [16]. Map-side joins produce final join results in the map

phase, and do not use the reduce phase. Because they do not need to pass intermediate results from map workers to reduce workers, map-side joins are more efficient than reduce-side joins; however, they can only be used in particular circumstances. Hadoop's map-side join [17], called the Map-Merge join [16], merges two input datasets that are partitioned and sorted on the join keys in advance, similar to merge join in traditional DBMS. However, all input datasets must be divided into the same number of partitions and must be sorted by the same key; otherwise, an additional job is required to bring the datasets under the condition. A broadcast join [5] distributes one of the input datasets to all map workers. It is efficient only if the size of the one dataset is small enough to be loaded into memory, or it may perform poorly because it has to read and process the dataset repeatedly.

Reduce-side joins can be used in more general cases, but are inefficient because large intermediate records are sent from map workers to reduce workers. Figure 1 illustrates the process followed by the basic reduce-side join algorithm, called the repartition join [5], with an example of a join between R(a,b) and S(a,c). In this example, all of the input records are sent to reduce workers to find records with the same join key, including redundant records marked with strikethrough text. Semijoin [5] in MapReduce works similarly to semijoin in traditional DBMS. It uses a three-step process: First, it finds unique join keys from an input dataset, say R. Second, it finds joined records in the other dataset, say S, with the unique join keys from the first step. Third, it produces final join results from the join between R and the joined records of S from the second step. Semijoin may reduce the size of intermediate results, but introduces additional I/O overhead because it runs each step in an independent MapReduce job. The results of its first two jobs are written to the underlying distributed file system and read in the next job. Our proposed approach operates in one MapReduce job. Map-Reduce-Merge [6] adds a merge phase after the reduce phase to support operations on multiple heterogeneous datasets; however, it does not reduce the size of the intermediate results.

Researchers have proposed approaches to optimize multi-way joins in MapReduce [18,19]. These approaches use a similar idea of minimizing the size of the replicated records that are sent to the reduce workers. We address only two-way joins in this paper, but our approach can be extended to multi-way joins by combining it with these approaches.

2.2 Join processing with bloom filters

A Bloom filter [8] is a probabilistic data structure that is used to test whether an element is a member of a set. It consists of an array of m bits and k independent hash functions. All the bits are initially set to zero. When an element is inserted, it hashes the element k times with k hash functions, and sets the positions in the array corresponding to the hash values to one. In a similar fashion, it checks whether an element is in the filter by hashing the element k times as in the insert operation, and checking whether all bits of the k hash positions are one.

Bloom filters have been used in join processing for the following reasons: (1) they may yield false positives, but do not yield false negatives. (2) They are space-efficient because of a fixed size, regardless of the number of inserted elements. A

join algorithm using the Bloom filter, called Bloomjoin [20], was proposed in the distributed database area. Bloomjoin creates a Bloom filter with the join keys of one relation, and filters out tuples of the other relation that are not matched by the Bloom filters. Bloomjoin can be combined with a group-by operation and extended to multi-way joins [21]. Some researchers optimize complex distributed multi-way joins using Bloom filters by determining the optimal join order and caching the filters [22,23]. However, they assume that the join relations are not split and not distributed to other nodes dynamically, unlike in a MapReduce environment.

Several recent studies have tried to adapt this algorithm to the MapReduce framework. A reduce-side join with a Bloom filter [11] was proposed, but it has a disadvantage in that input datasets have to be processed multiple times because it creates the filter via an independent MapReduce job. Zhang et al. [12] extended this approach to multi-way joins, but their work has the same disadvantage. Koutris [9] theoretically investigated join techniques using Bloom filters within a single MapReduce job, but did not provide specific technical details. Lee et al. [10] addressed implementation issues and provided a framework for join processing using Bloom filters within a single MapReduce job. Though these approaches apply Bloom filters in slightly different ways, they all apply the filters without regard to performance. We use the filters based on a threshold for stable performance according to their FPRs.

3 TMFR-Join

This section explains our join processing method with threshold-based filtering in MapReduce, TMFR-Join. As we have implemented it into Hadoop [4], an open-source implementation of MapReduce, we will use Hadoop terminology in the remainder of this paper. A Hadoop cluster is composed of one jobtracker node and a number of tasktracker nodes. The jobtracker checks the statuses and controls the actions of tasktrackers through heartbeat messages. Tasktrackers run one or more mapper and reducer processes, which execute map and reduce tasks, respectively, according to the configuration.

Note that this is an extension of our previous work, which applies Bloom filters to Hadoop regardless of performance [10]. In the previous work, two major changes were made to Hadoop for application of filters. First, map tasks are scheduled according to the processing order of the input datasets. Within an input dataset, map tasks are scheduled in the order of the input split size, as in the original Hadoop. Second, filters are constructed dynamically in a single MapReduce job. We add the filter estimation phase for application of filters based on a user-configured threshold to this work.

3.1 Execution overview

Let us first look at the overall flow of execution of TMFR-Join. Figure 2 depicts an example of a join between two datasets, R and S, in TMFR-Join. In this example, we suppose that R is chosen to be processed first; that is, filters are built on R. When a user runs a MapReduce program, the following sequence of actions is performed.

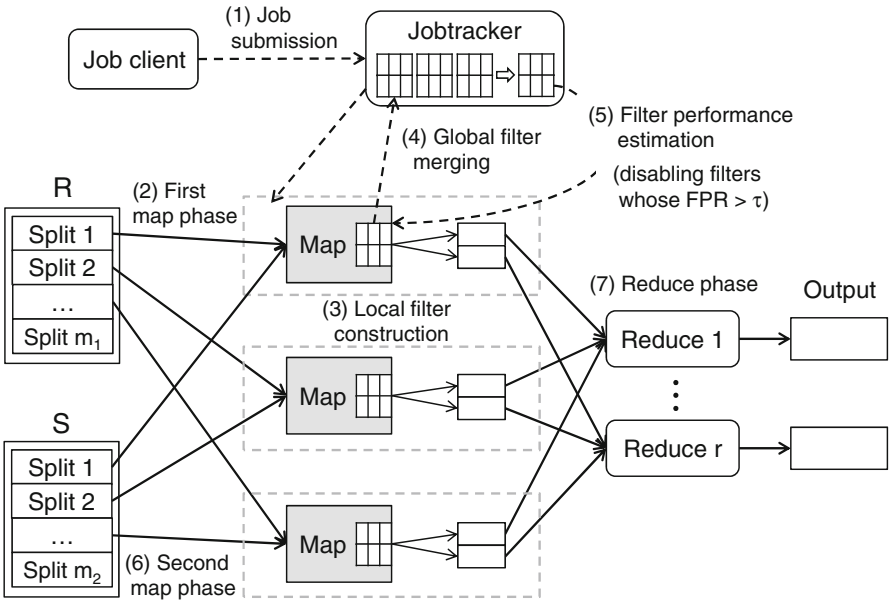

Fig. 2 Execution overview

1. *Job submission.* On submission of a MapReduce job, 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 the necessary information to run on a tasktracker, such as job configuration and location of the corresponding input/output files. The job configuration includes additional filter information such as parameter types and values for the filter to use.
2. *First map phase.* The jobtracker assigns the m_1 map tasks for R or the reduce tasks to those tasktrackers that have idle mappers or reducers. A mapper 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 mappers are divided into r partitions, which are sent to their corresponding reducers. For each partition, a specified type of filter is created by inserting the keys of its intermediate pairs. These filters are called local filters, because they are built for the intermediate results in a single tasktracker. Each tasktracker merges the individual filters from each map task and maintains only r filters, until all m_1 map tasks for the first input dataset R are completed.
4. *Global filter merging.* When all the m_1 map tasks have been completed, the jobtracker stops assigning map tasks and requests that all tasktrackers send it their local filters via heartbeat responses. The jobtracker then merges all the local filters to construct the global filters for R, which contain all the join keys that are processed in all the tasktrackers.
5. *Filter performance estimation.* The jobtracker estimates the performance of the merged global filters by estimating their FPRs. The estimation method depends

on the filtering techniques that are applied; we describe this issue in Sect. 3.2. Assuming that the FPRs can be estimated, filters with FPRs exceeding the given threshold τ are disabled. The jobtracker then sends the global filters to all the tasktrackers.

6. *Second map phase.* The jobtracker assigns the m_2 map tasks for S or the remaining reduce tasks to the tasktrackers. Mappers run the assigned tasks with the received global filters, and intermediate pairs with keys that are not contained in the global filters are filtered out.
7. *Reduce phase.* This step is the same as the reduce phase in Hadoop. Reducers read the corresponding intermediate pairs from all mappers using remote procedure calls. Each reducer then sorts the intermediate pairs and runs the reduce function. Final output results are then written to the given output path.

3.2 Filtering techniques applicable to TMFR-Join

Efficient filtering techniques for joins depend on the distribution of the join keys and the number of records joined. For this reason, we designed our framework to apply various types of filters. Any filtering techniques can be plugged into TMFR-Join as long as they support the following operations:

- `insert(key)`: insertion of the specified `key` into the filter.
- `contains(key)`: returning whether it contains the specified `key` or not.
- `merge(filter)`: merging with another `filter` of the same type.
- `estimateFPR()`: computation of filter's FPR.

The `insert` and `contains` operations are the basic operations for testing (approximate) membership. TMFR-Join additionally requires `merge` and `estimateFPR` operations for the filters.

Some filtering techniques, such as Bloom filter [8] and its variants [13], Interval filter [15], and Quotient filter [14], already support the operations. Table 1 shows `merge` and `estimateFPR` operations for the example filters. The operations will be naturally different for each filtering technique. The Bloom filter operates as described in Sect. 2.2. Its variants that generate no false negatives can also be used. The Interval filter uses an array of m bits, and its lower bound lb and upper bound ub should be set. Without loss of generality, suppose that the elements to be inserted into the filter are integers. Then, the range [lb, ub] is split into m intervals, and each interval has a length itv of $(\text{ub} - \text{lb})/m$. Each bit in the array represents an interval, and the interval bit for an element with value v is the bit of the position $(v - \text{lb})/\text{itv}$. The filter inserts an

Table 1 `Merge` and `estimateFPR` operations for some example filters

	`merge`	`estimateFPR`
Bloom filter	Bitwise OR	$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k$
Interval filter	Bitwise OR	$\left(1 - \left(1 - \frac{1}{m}\right)^{n}\right)$
Quotient filter	Similar to merge sort in DBMS	$\left(1 - \left(1 - \frac{1}{2^{(q+r)}}\right)^{n}\right)$

Table 2 Global filter merging time

Number of tasktrackers	3	5	7	10
Merging time (s)	22.137	26.735	25.012	28.887

element by setting the value of its interval bit to one, and checks whether an element is in the filter by checking its interval bit. The Quotient filter stores a p-bit fingerprint for each element. It consists of an array of size 2^q of $(r + 3)$ items, where $p = q + r$. (We here omit the details of its insert and contains operations. See the algorithms in the original paper [14].)

To compute the FPRs of the example filters, we need to know the number of inserted elements n in the merged filters. However, keeping track of all the distinct values inserted into each local filter to compute n is impractical. We only know the number of true bits, not that of inserted elements, in the merged filters. Therefore, we estimate the number of inserted elements with the expected number of true bits in the filters after n elements are inserted [24]. Let p_0 be the probability of a bit being false after n elements are inserted for a filtering technique. For example, p_0 is $\left(1 - \frac{1}{m}\right)^n$ for the Interval filter and $\left(1 - \frac{1}{m}\right)^{kn}$ for the Bloom filter because it uses k hash functions. Thus, the probability of a bit being true is $(1 - p_0)$. Therefore, the expected number of true bits can be computed by multiplying it by the total number of bits for a filter.

Global filters are constructed by performing the merge operation repeatedly. It should be noted that the global filter construction is expected to take a long time if the number of reduce tasks or tasktrackers is large, because it is currently performed by the jobtracker alone. Table 2 shows the time elapsed for the global filter merging with 4 Mb Bloom filters on an 11-node cluster, varying the number of tasktrackers. The details of the experimental environment can be found in Sect. 5.1. The time for merging filters tends to increase as the number of nodes becomes larger, and it also depends on the difference between the time that the first node finishes its assigned map tasks for the build input, and the time that the last node finishes its corresponding map tasks. This overhead could be distributed by merging local filters hierarchically, although this has not yet been implemented.

3.3 Early detection of FPR threshold being exceeded

The execution flow described in Sect. 3.1 estimates the FPR after the merging of all local filters into global filters. Before the estimation, all mappers insert the join keys of the first input dataset into the filters, which incurs CPU overhead. Furthermore, during the global filter merging, network I/O costs associated with communicating the filters between the jobtracker and tasktrackers are incurred. These costs can be reduced if we find that the FPRs of global filters are greater than the given threshold in advance. The earlier the situation is detected, the more costs can be reduced.

Before merging local filters into global filters, only the FPRs of the individual local filters in each tasktracker can be computed. However, if the FPRs of the global filters can be estimated along with them, then the situation could be detected much earlier. Consequently, we add an optional operation, `estimateUnionFPR`, to the filtering techniques. If this operation is supported, the jobtracker can estimate the

FPRs of the global filters as follows: in the first map phase, each tasktracker sends the jobtracker only the FPR values of its local filters, not the whole local filters themselves, via heartbeat messages when a map task is complete and its FPRs are changed. The jobtracker then estimates the FPRs of the global filters using the operation.

The FPRs of the global filters can be estimated with those of the individual local filters using methods similar to those used by Michael et al. [22]. Let the FPRs of two local filters be $P(A)$ and $P(B)$. The FPR of the merged filter $P(A \cup B)$ can be computed as $P(A) + P(B) - P(A \cap B)$. Assuming that the distribution of data is independent, $P(A \cap B) = P(A) \cdot P(B)$, we can estimate $P(A \cap B)$ with individual FPRs $P(A)$ and $P(B)$.

We designed the signature of the operation as `estimateUnionFPR (`$\texttt{FPRs}_{\text{global}}$, $\texttt{FPRs}_{\text{local\_prev}}$, $\texttt{FPRs}_{\text{local\_cur}}$). Note that both the previous and current FPRs of the local filters are required. The jobtracker should repeatedly merge the local filters from a tasktracker with their updated FPRs. However, estimating the FPRs of the merged filters may not be idempotent. If it merges the same local filters multiple times, its FPRs will be changed. In this case, the `estimateUnionFPR` operation must first compute the FPRs of the global filters, excluding the previous FPRs of the local filters, and then estimate the new FPRs of the global filters by merging the current FPRs of the local filters. For example, suppose that the current estimated FPRs of a local filter and the corresponding global filter are 0.4 and 0.2, respectively. If the next FPR of the local filter is 0.3, the FPR of the global filter with the exception of the previous FPR is estimated to be 0.25 by calculating the expression $0.4 = P(G) + 0.2 - 0.2 \cdot P(G)$. Then, the current FPR of the global filter, including the new FPR of the local filter, is estimated to be 0.475, which is $0.25 + 0.3 - 0.25 \cdot 0.3$ This is possible if the operation is commutative. The order in which the filters are merged does not affect the estimation results.

4 Cost and FPR threshold analysis

This section addresses how to choose an FPR threshold value. We first define the notion of an equilibrium FPR threshold, and explain how to compute it using a cost model, which is a variant of Herodotou's Hadoop performance model [25].

4.1 Equilibrium FPR threshold

The aim of TMFR-Join is to guarantee join performance that is close to the better of the existing join processing techniques, both with and without filters, by disabling the filters whose estimated FPRs are greater than a user-configured threshold. Hence, the best FPR threshold value must be the FPR when the costs in both cases are the same, because the filters should be applied only if they are beneficial. We define the value as equilibrium FPR threshold τ_{eq}. If an FPR threshold is set as a value larger than τ_{eq}, it may result in the filters being used in cases where intermediate records are not filtered out enough to improve the execution time. Conversely, if the threshold is set as a value smaller than τ_{eq}, it may result in the filters being disabled in cases where intermediate records are filtered out enough.

To compute the equilibrium FPR threshold, we need to compute the join costs with and without filters. Let T_{job} be the total cost of a MapReduce job, T_{map} be the cost of all map tasks for the job, T_{filter} be the cost of merging filters locally and globally, and T_{reduce} be the cost of all reduce tasks for the job. Consequently, the total cost of a MapReduce job T_{job} can be expressed as $T_{\text{map}} + T_{\text{filter}} + T_{\text{reduce}}$. When execution costs with and without filters are different from each other, we use superscripts f and h to denote the costs of join processing with and without them, respectively. Accordingly, the total cost of a MapReduce job in both of them can be expressed as follows:

$$T^{\text{f}}_{\text{job}} = T^{\text{f}}_{\text{map}} + T_{\text{filter}} + T^{\text{f}}_{\text{reduce}}$$
$$T^{\text{h}}_{\text{job}} = T^{\text{h}}_{\text{map}} + T^{\text{h}}_{\text{reduce}}$$

Let D_{job} denote the difference of the total cost between join processing with and without filters. Then, τ_{eq} can be obtained by finding the FPR value to make D_{job} to zero.

$$D_{\text{job}} = T^{\text{f}}_{\text{job}} - T^{\text{h}}_{\text{job}} = (T^{\text{f}}_{\text{map}} - T^{\text{h}}_{\text{map}}) + T_{\text{filter}} + (T^{\text{f}}_{\text{reduce}} - T^{\text{h}}_{\text{reduce}}) \quad (1)$$

We examine the cost differences in detail in the following subsection.

4.2 Cost model

Herodotou's Hadoop performance model [25] was chosen to compute an FPR threshold value because it describes the execution of a MapReduce job on Hadoop in detail. It was slightly adjusted to include features for filtering. Figure 3 shows the process of execution of a map and reduce task in TMFR-Join, illustrating the shuffle phase in detail. Note that we use the term shuffle phase for the whole process, from the point where map tasks produce intermediate records to the point where reduce tasks consume them, as described by Tom White [17]. This process is virtually the same as

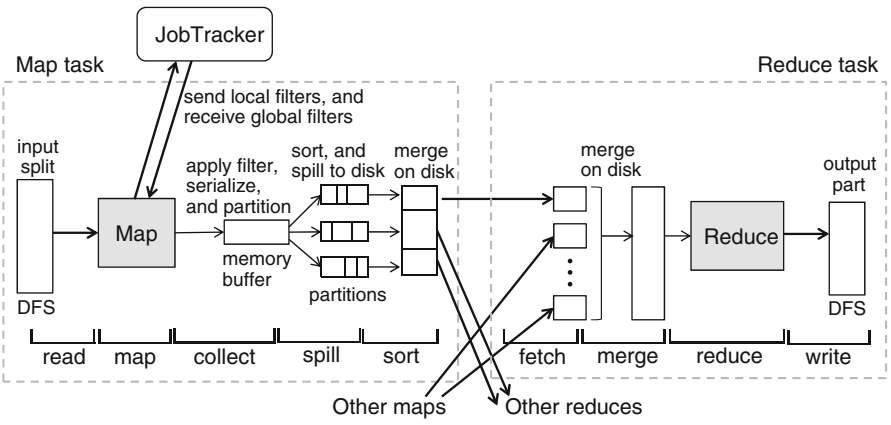

Fig. 3 Shuffle phase in MapReduce

Table 3 System parameters

Parameter	Definition
$c_{\text{r\_dfs}}$	I/O cost of reading from distributed file system per byte
$c_{\text{w\_dfs}}$	I/O cost of writing to distributed file system per byte
$c_{\text{r\_loc}}$	I/O cost of reading from local disk per byte
$c_{\text{w\_loc}}$	I/O cost of writing to local disk per byte
c_{tr}	Network cost of transferring data per byte
$c_{\text{c\_map}}$	CPU cost of executing map function per record
$c_{\text{c\_red}}$	CPU cost of executing reduce function per record
$c_{\text{c\_part}}$	CPU cost of partitioning and (de)serializing per record
$c_{\text{c\_sort}}$	CPU cost of sorting per record
$c_{\text{c\_merge}}$	CPU cost of merging sorted data per record
$c_{\text{c\_fltr}}$	CPU cost of inserting an element into filters or checking that an element is in the filters per record
$c_{\text{c\_union}}$	CPU cost of merging a filter

Table 4 Hadoop parameters

Parameter	Definition
$\#_{\text{nodes}}$	Number of tasktracker nodes
$\#_{\text{map}}$	Number of map tasks for a job (i.e., $\#_{\text{map}} = \#_{\text{map}_r} + \#_{\text{map}_s}$)
$\#_{\text{reduce}}$	Number of reduce tasks for a job
$\#_{\text{map/node}}$	Maximum number of map tasks that can be simultaneously run by a tasktracker

Table 5 Data parameters

Parameter	Definition
$b_{\text{r}}, b_{\text{s}}$	Size in bytes of input datasets R and S, respectively
$l_{\text{r}}, l_{\text{s}}$	Length of a record in input datasets R and S, respectively
$n_{\text{r}}, n_{\text{s}}$	Number of records in input datasets R and S, respectively (i.e., $n_{\text{r}} = b_{\text{r}}/l_{\text{r}}, n_{\text{s}} = b_{\text{s}}/l_{\text{s}}$)
b_{f}	Size in bytes of a filter

that in the original Hadoop except that the operations for filtering are carried out in the collect stage in map tasks. It should be noted that the filter merging processes are not included because it is from the point of view of a single map task. It is included when we consider the execution of all the map tasks.

We assume that we have a MapReduce job for a join on datasets R and S, and that the parameters in Tables 3, 4 and 5 are given. Note that the parameters were simplified in comparison with the original model, and only $c_{\text{c\_fltr}}$, $c_{\text{c\_union}}$, and b_{f} were added to consider the cost of filtering. The system parameters in Table 3, which relate to I/O and CPU cost, can be defined in terms of time per byte according to the hardware configuration. The parameters in Table 4 can be obtained from the Hadoop configuration. We believe that the data parameters in Table 5 can be defined depending on the application, which knows the properties of data to analyze. Otherwise, these parameters may be given by users.

We also assume that the combiner and compression features are not used, for simplicity. Since Herodotou's cost model [25] considers these features, our cost model can be extended as in the original. Next, we examine the execution costs stage by stage, focusing on the cost differences between join processing with and without filters.

4.2.1 Map task

The execution of a map task can be divided into five stages, as shown in Fig. 3. The total cost of all map tasks T_{map} can be computed as the sum of the costs during the five stages.

$$T_{\text{map}} = C_{\text{read}} + C_{\text{map}} + C_{\text{collect}} + C_{\text{spill}} + C_{\text{sort}}$$

Read and map stages Each map task reads the corresponding input split and converts each record to a key/value pair. It then executes the map function and produces intermediate map output records. Hence, the costs during read and map stages can be computed as follows:

$$C_{\text{read}} = (b_{\text{r}} + b_{\text{s}}) \cdot c_{\text{r\_dfs}}$$
$$C_{\text{map}} = (n_{\text{r}} + n_{\text{s}}) \cdot c_{\text{c\_map}}$$

The costs are the same in both join processing with and without filters, so these stages do not need to be considered in determining an FPR threshold.

Collect stage All the map output records are partitioned, and processed with an `insert` or `contains` operation if filters are used. The map output records from the first input dataset, say R, are inserted into local filters; or, those from the second input dataset, say S, are checked to determine whether they are in the global filters. The map output records that are not filtered out are collected into a memory buffer.

The costs of the collect stage in join processing with and without filters are

$$C^{\text{f}}_{\text{collect}} = (n_{\text{r}} + n_{\text{s}}) \cdot (c_{\text{c\_part}} + c_{\text{c\_fltr}})$$
$$C^{\text{h}}_{\text{collect}} = (n_{\text{r}} + n_{\text{s}}) \cdot c_{\text{c\_part}}$$

Note that the additional CPU cost of executing filter operations results from applying the filters.

The number of intermediate map output records n_{inter} will differ as well.

$$n^{\text{f}}_{\text{inter}} = n_{\text{r}} + n_{\text{s}} \cdot \sigma_{\text{s\_r}} + n_{\text{s}} \cdot (1 - \sigma_{\text{s\_r}}) \cdot p$$
$$n^{\text{h}}_{\text{inter}} = n_{\text{r}} + n_{\text{s}}$$

where $\sigma_{\text{s\_r}}$ is the ratio of the joined records of S with R, and p is the actual FPR of the global filters. n_{r} signifies the number of first input dataset records that are not filtered out and used to create filters. $n_{\text{s}} \cdot \sigma_{\text{s\_r}}$ signifies the number of second input dataset records that are joined, and $n_{\text{s}} \cdot (1 - \sigma_{\text{s\_r}}) \cdot$ signifies the number of second input dataset

records that are not joined but passed to reducers as false positives. Without filters, n_{inter} is equal to $n_{\text{r}} + n_{\text{s}}$ because input records are not filtered out at all.

Spill stage When the data size in the buffer reaches a given threshold, the data partitions are sorted and written to local disk. The cost of the spill stage depends on the number of spills, and is determined according to the size of the spill buffer, which is configured via Hadoop parameters such as `io.sort.mb`, `io.sort.spill.percent`, and `io.sort.record.percent`. Let b_{buf} be the buffer size in bytes, and let the length of an intermediate record l_{inter} be $(l_{\text{r}} + l_{\text{s}})$, assuming both the records from R and S are joined without projection. Accordingly, the number of records that can be included in buffer n_{buf} can be expressed as $b_{\text{buf}} / l_{\text{inter}}$. Then, the number of spills that are performed in all map tasks $\#_{\text{spill}}$ can be estimated as follows:

$$\#_{\text{spill}} = \left\lceil \frac{n_{\text{inter}} \cdot l_{\text{inter}}}{b_{\text{buf}}} \right\rceil = \left\lceil \frac{n_{\text{inter}}}{n_{\text{buf}}} \right\rceil$$

With the estimate, the cost of the spill stage is computed as follows:

$$C_{\text{spill}} = \#_{\text{spill}} \cdot \left(n_{\text{buf}} \cdot \log_2 \left(\frac{n_{\text{buf}}}{\#_{\text{reduce}}} \right) \cdot c_{\text{c\_sort}} + b_{\text{buf}} \cdot c_{\text{w\_loc}} \right) \tag{2}$$

where $n_{\text{buf}} \cdot \log_2 \left(\frac{n_{\text{buf}}}{\#_{\text{reduce}}} \right)$ represents the CPU cost of sorting each partition on average, and $b_{\text{buf}} \cdot c_{\text{w\_loc}}$ represents the I/O cost of writing the data in the buffer to local disk. Although Eq. (2) can be used to compute the costs of the spill stage in both join processing with and without filters, the costs may vary because the numbers of intermediate records n_{inter} may be different in each case.

Sort stage In the sort stage, the spilled partitions are merged into a single file. This stage performs an external merge sort similar to the merge stage in a reduce task. The merging process may be performed in multiple merge passes according to the number of spills. The total number of merge passes depends on the number of spills and the number of spills to merge at once, which is configured by `io.sort.factor` and denoted as nf. We compute the number of spills in all map tasks $\#_{\text{spill}}$ in the spill stage, but each map task only merges its own spills. Assuming that each map task merges the same number of spills, the number of spills that are merged in each map task can be calculated using $\#_{\text{spill}}/\#_{\text{map}}$, and the number of merge passes $\#_{\text{merge}}$ can be expressed as $\lceil \log_{\text{nf}} (\#_{\text{spill}}/\#_{\text{map}}) \rceil$. Let us suppose that Hadoop reads all the spills and writes their intermediate merge output in each merge pass, although Hadoop does not always need to merge all spills. Then, the cost of the sort stage can be computed as follows:

$$\begin{aligned} C_{\text{sort}} &= \#_{\text{merge}} \cdot \#_{\text{map}} \cdot \left(\frac{\#_{\text{spill}}}{\#_{\text{map}}} \cdot b_{\text{buf}} \cdot (\text{c}_{\text{r\_loc}} + \text{c}_{\text{w\_loc}}) + \frac{\#_{\text{spill}}}{\#_{\text{map}}} \cdot \text{n}_{\text{buf}} \cdot \text{c}_{\text{c\_merge}} \right) \\ &= \#_{\text{merge}} \cdot \#_{\text{spill}} \cdot (\text{b}_{\text{buf}} \cdot (\text{c}_{\text{r\_loc}} + \text{c}_{\text{w\_loc}}) + \text{n}_{\text{buf}} \cdot \text{c}_{\text{c\_merge}}) \end{aligned}$$

As in the spill stage, the cost of the sort stage in join processing with and without filters may vary according to the number of intermediate records.

4.2.2 Filter merging

Additional costs arise during the filter merging process. When tasktrackers process a map task for the first input dataset, one filter is created per partition, that is, per reducer. The number of filters that are created in each map task is equal to the number of reducers $\#_{\text{reduce}}$. Each tasktracker locally merges the filters from its own map tasks, so it maintains only $\#_{\text{reduce}}$ filters. The filters are merged globally after all map tasks for the first input dataset have been completed. During this process, tasktrackers wait for the merged filters and do not run other map tasks. This loss must also be taken into account.

Consequently, the total cost for merging filters T_{filter} can be divided into three parts: the cost of merging locally in each tasktracker $C_{\text{filter\_local}}$, the cost of merging globally in the jobtracker $C_{\text{filter\_global}}$, and the cost of waiting for the global filters (without tasktrackers running other map tasks) $C_{\text{filter\_wait}}$. T_{filter} can be computed as follows:

$$T_{\text{filter}} = C_{\text{filter\_local}} + C_{\text{filter\_global}} + C_{\text{filter\_wait}}$$

where

$$\begin{aligned} C_{\text{filter\_local}} &= \#_{\text{map}} \cdot \#_{\text{reduce}} \cdot c_{\text{c\_union}} \\ C_{\text{filter\_global}} &= \#_{\text{nodes}} \cdot \#_{\text{reduce}} \cdot (2 \cdot c_{\text{tr}} + c_{\text{c\_union}}) \\ C_{\text{filter\_wait}} &= \#_{\text{nodes}} \cdot \#_{\text{map/node}} \cdot \frac{T_{\text{map}}}{\#_{\text{map}}} \end{aligned}$$

$\#_{\text{reduce}}$ signifies the number of filters, and $C_{\text{filter\_local}}$ signifies the CPU cost for merging filters for all map tasks. $2 \cdot c_{\text{tr}}$ signifies the network cost of communicating the filters between a tasktracker and the jobtracker. $C_{\text{filter\_global}}$ includes the network and CPU cost incurred by the jobtracker merging filters from all the tasktrackers. The waiting cost during the global filter merging $C_{\text{filter\_wait}}$ cannot be measured consistently, because it is affected by straggler nodes, input data skew, node capability, and so on. We approximate it as the cost for running the maximum number of map tasks simultaneously on all tasktrackers, assuming the difference between the completion times of the first and last nodes that finish their map tasks is smaller than the time to run a map task.

4.2.3 Reduce task

The execution of a reduce task can be divided into the four stages shown in Fig. 3. Note that we renamed the first stage of the reduce tasks to Fetch, whose original name was Shuffle in [25], because we use the term shuffle phase in a broad sense. The total cost of reduce tasks T_{reduce} can be computed as the sum of the costs during the four stages of the reduce task.

$$T_{\text{reduce}} = C_{\text{fetch}} + C_{\text{merge}} + C_{\text{reduce}} + C_{\text{write}}$$

Fetch stage In the fetch stage, the intermediate records produced from map tasks are copied from mappers to reducers. Suppose that no merging process occurs in this stage. Then, the total cost of the fetch stage is

$$C_{\text{fetch}} = n_{\text{inter}} \cdot l_{\text{inter}} \cdot c_{\text{tr}}$$

Merge stage This stage merges the sorted partitions that are fetched from mappers. It works similar to the sort stage in map task but the number of merge passes $\#_{\text{merge}}$ is switched to $\lceil \log_f (\#_{\text{map}}) \rceil$, because each reduce task merges $\#_{\text{map}}$ partitions that came from mappers. Suppose that all the partitions reside on disk. Then, the total cost of the merge phase can be computed as follows:

$$C_{\text{merge}} = \#_{\text{merge}} \cdot n_{\text{inter}} \cdot (l_{\text{inter}} \cdot (c_{\text{r\_loc}} + c_{\text{w\_loc}}) + c_{\text{c\_merge}})$$

Reduce stage The merged data are processed with a given reduce function. The total cost of the reduce stage is

$$C_{\text{reduce}} = n_{\text{inter}} \cdot (l_{\text{inter}} \cdot c_{\text{r\_loc}} + c_{\text{c\_red}})$$

It is clear that the costs of the above three stages depend on the number of intermediate records. Therefore, the costs during each stage in join processing with and without filters may be different.

Write stage The final results of the job are written to the distributed file system. Let b_{out} be the size (in bytes) of the final results. Then, the costs of the reduce and write stages can be computed as follows:

$$C_{\text{write}} = b_{\text{out}} \cdot c_{\text{w\_dfs}}$$

Although we may not know the size of the final results in advance, the cost of the write stage is the same regardless of join techniques as long as they produce correct join results. Therefore, we can omit the write stage from cost estimation.

4.3 Determining FPR threshold

The equilibrium FPR threshold τ_{eq} can be computed based on the cost model, which we examined step by step in Sect. 4.2. Recall that the difference of the total cost between join processing with and without filters D_{job} can be computed as the sum of the cost differences in each phase. Equation (1) can be rewritten as follows:

$$\begin{aligned} D_{\text{job}} &= T^{\text{f}}_{\text{job}} - T^{\text{h}}_{\text{job}} = (T^{\text{f}}_{\text{map}} - T^{\text{h}}_{\text{map}}) + T_{\text{filter}} + (T^{\text{f}}_{\text{reduce}} - T^{\text{h}}_{\text{reduce}}) \\ &= (D_{\text{collect}} + D_{\text{spill}} + D_{\text{sort}}) + T_{\text{filter}} + (D_{\text{fetch}} + D_{\text{merge}} + D_{\text{reduce}}) \end{aligned}$$

where D_{stage} represents the cost difference in the corresponding stage.

The cost difference in the collect stage D_{collect} can be computed as follows:

$$D_{\text{collect}} = C^{\text{f}}_{\text{collect}} - C^{\text{h}}_{\text{collect}} = (n_{\text{r}} + n_{\text{s}}) \cdot c_{\text{c\_fltr}}$$

The difference of the intermediate records d_{inter} after the collect phase is given as follows:

$$\begin{aligned} d_{\text{inter}} &= n^{\text{f}}_{\text{inter}} - n^{\text{h}}_{\text{inter}} \\ &= (n_{\text{r}} + n_{\text{s}} \cdot \sigma_{\text{s\_r}} + n_{\text{s}} \cdot (1 - \sigma_{\text{s\_r}}) \cdot p) - (n_{\text{r}} + n_{\text{s}}) \\ &= -n_{\text{s}}(1 - \sigma_{\text{s\_r}})(1 - p) \end{aligned} \tag{3}$$

Since $0 \leq \sigma_{\text{s\_r}} \leq 1$ and $0 \leq p \leq 1$, unless $\sigma_{\text{s\_r}}$ is equal to one, the number of intermediate records n_{inter} may be reduced according to the FPR p, at the cost of applying filter operations. Accordingly, the difference in the number of spills $d_{\#_{\text{spill}}}$ in the spill and sort stage is

$$\begin{aligned} d_{\#_{\text{spill}}} &= \#^{\text{f}}_{\text{spill}} - \#^{\text{h}}_{\text{spill}} = \left\lceil \frac{n^{\text{f}}_{\text{inter}}}{n_{\text{buf}}} \right\rceil - \left\lceil \frac{n^{\text{h}}_{\text{inter}}}{n_{\text{buf}}} \right\rceil \\ &= \left\lceil \frac{(n_{\text{r}} + n_{\text{s}} \cdot \sigma_{\text{s\_r}} + n_{\text{s}} \cdot (1 - \sigma_{\text{s\_r}}) \cdot p)}{n_{\text{buf}}} \right\rceil - \left\lceil \frac{n_{\text{r}} + n_{\text{s}}}{n_{\text{buf}}} \right\rceil \\ &= \begin{cases} -\left\lceil \frac{n_{\text{s}}(1-\sigma_{\text{s\_r}})(1-p)}{n_{\text{buf}}} \right\rceil, & \text{if } n_{\text{s}}(1 - \sigma_{\text{s\_r}})(1 - p) \bmod n_{\text{buf}} \\ & \quad > (n_{\text{r}} + n_{\text{s}}) \bmod n_{\text{buf}} \\ -\left\lfloor \frac{n_{\text{s}}(1-\sigma_{\text{s\_r}})(1-p)}{n_{\text{buf}}} \right\rfloor, & \text{otherwise} \end{cases} \end{aligned}$$

Hence, the cost differences in the spill and sort stage can be expressed as follows, assuming that the numbers of merge passes for both join processing with and without filters are the same.

$$\begin{aligned} D_{\text{spill}} &= d_{\#_{\text{spill}}} \cdot \left(n_{\text{buf}} \cdot \log_2 \left(\frac{n_{\text{buf}}}{\#_{\text{reduce}}} \right) \cdot c_{\text{c\_sort}} + b_{\text{buf}} \cdot c_{\text{w\_loc}} \right) \\ D_{\text{sort}} &= d_{\#_{\text{spill}}} \cdot \#_{\text{merge}} \cdot (b_{\text{buf}} \cdot (c_{\text{r\_loc}} + c_{\text{w\_loc}}) + n_{\text{buf}} \cdot c_{\text{c\_merge}}) \end{aligned}$$

Both stages' cost differences are affected by the number of intermediate records.

The cost differences of the fetch, merge, and reduce stage in the reduce tasks are

$$\begin{aligned} D_{\text{fetch}} &= d_{\text{inter}} \cdot l_{\text{inter}} \cdot c_{\text{tr}} \\ D_{\text{merge}} &= d_{\text{inter}} \cdot \#_{\text{merge}} \cdot (l_{\text{inter}} \cdot (c_{\text{r\_loc}} + c_{\text{w\_loc}}) + c_{\text{c\_merge}}) \\ D_{\text{reduce}} &= d_{\text{inter}} \cdot (l_{\text{inter}} \cdot c_{\text{r\_loc}} + c_{\text{c\_red}}) \end{aligned}$$

These costs also depend on the number of intermediate records.

Consequently, all of the cost factors depend on the number of intermediate results. According to Eq. (3), we can further infer that the total cost is eventually determined by two factors: the ratio of the joined records and the FPRs of the filters. The other

parameters can be regarded as constants. Thus, τ_{eq} can be precomputed if the ratio of the joined records is known.

If the ratio of the joined records is not known in advance, the FPR threshold can be configured not as τ_{eq} but as the FPR value computed with the maximum ratio of the joined records, which is known or given by users. We suggest choosing the maximum ratio of the joined records as a large enough value, in case the filters are not beneficial. The FPR threshold value corresponding to the ratio would then be small enough. If the FPR threshold is small, the opportunity to improve performance with filters is reduced. For example, filters may not be applied if the join ratio is much smaller than the maximum ratio and the actual FPRs are slightly larger than the threshold value. Despite this, we believe that an ample maximum ratio should be used to prevent join performance degradation.

5 Performance evaluation

This section presents our experimental results. We implemented the proposed framework on Hadoop 0.20.2. In addition, we implemented three example filtering techniques, the Bloom filter [8], Interval filter [15], and Quotient filter [14], to show that our method works with various filtering techniques.

5.1 Experimental setup

All the experiments were run on a cluster of 11 machines consisting of one jobtracker and 10 tasktrackers. Each machine comprised a 3.1 GHz quad-core CPU, 4 GB RAM, and a 2 TB hard disk. The operating system was 32-bit Ubuntu 10.10, and the Java version used was 1.6.0_26.

We configured Hadoop based on the real-world cluster configuration parameters in the Hadoop official documentation [26]. We set the HDFS block size to 128 MB and the replication factor to three. 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.

We used TPC-H benchmark [27] datasets, which are widely accepted in the industry, with scale factor 100. The scale factor is the total 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. Therefore, we added some selection predicates to control the ratio of joined records. Our test query can be expressed in SQL-like syntax as follows:

```
SELECT substr(l.*, 0, l_lineitem), substr(o.*, 0, l_orders)
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 0.8. We also set the query

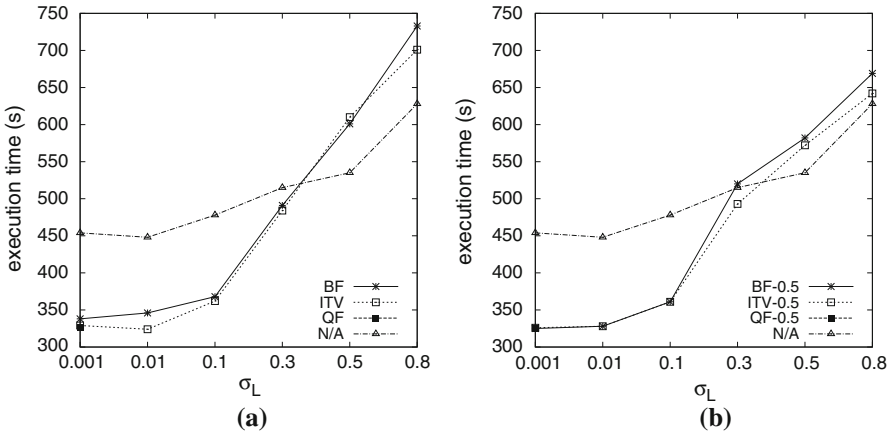

Fig. 4 Execution times with various filtering techniques. **a** MFR-Join (with no threshold), **b** TMFR-Join ($\tau = 0.5$)

results as substrings of the joined records in each table. l_{lineitem} and l_{orders} are the lengths of the substrings, so the length of an intermediate record is $l_{\text{lineitem}} + l_{\text{orders}}$. As the length increased, so did the filtering performance. We set the lengths for both tables to 10, assuming the case of a projection query. The Hadoop program for the test query was hand-coded, and we chose `orders` as the first input and `lineitem` as the second input.

5.2 Experimental results

We compared the performance of our method to that of the existing repartition join [5], which does not use filters, and join processing with filters (MFR-Join) [10]. Three types of filters, Bloom filter (BF), Interval filter (ITV), and Quotient filter (QF), were applied, and the parameters for each filter were adjusted to set the size of a filter to 4 Mb.

First, we ran the test query while varying the ratio of the joined records. The FPR threshold for TMFR-Join was set to 0.5, which shows the best results in our experiments. Figure 4 shows the execution times of the test query using each join technique. With the repartition join (N/A) as a baseline, Fig. 4a and b shows the results of join processing using filters without and with threshold-based filtering, respectively. As shown in Fig. 4a, the performance of the existing MFR-Join with no threshold decreases as more records participate in the join, because the number of redundant records that can be filtered out is reduced. On the other hand, in Fig. 4b, TMFR-Join exhibits stable performance close to that of the repartition join (N/A) even when σ_{L} is greater than 0.3, regardless of the filtering technique used. Note that joins with Quotient filters did not finish when σ_{L} was greater than or equal to 0.01, because they cannot contain more than their size of elements, and their merge operations have a time complexity on the order of the number of inserted elements.

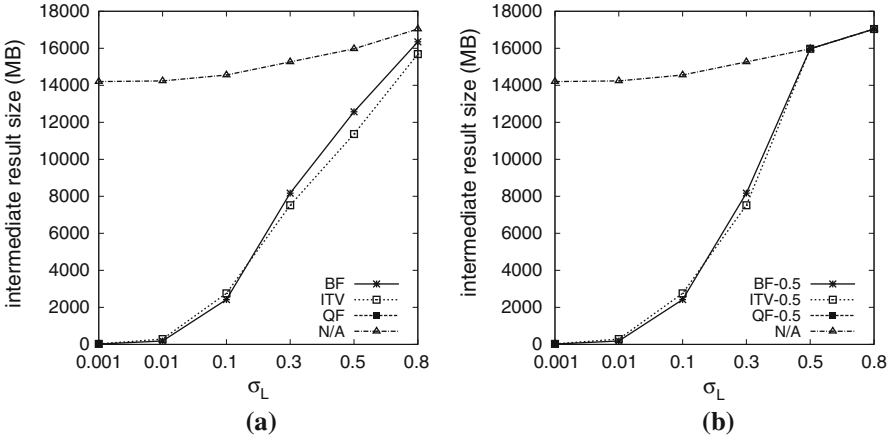

Fig. 5 Intermediate result sizes with various filtering techniques.**a** MFR-Join (with no threshold), **b** TMFR-Join ($\tau = 0.5$)

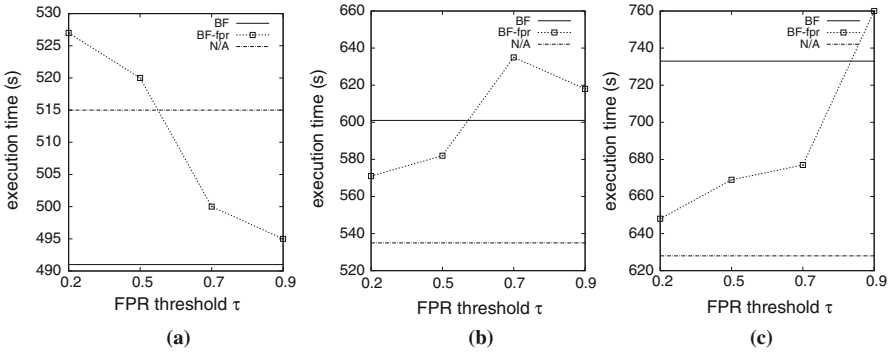

Fig. 6 Execution times of TMFR-Join with bloom filters varying FPR thresholds. **a** $\sigma_L = 0.3$, **b** $\sigma_L = 0.5$, **c** $\sigma_L = 0.8$

Figure 5 shows the intermediate result sizes in each case. In Fig. 5a, the repartition join is the largest in size because it emits all input records as intermediate results. When σ_L is small, a large number of redundant records are filtered out; whereas, when σ_L is large, the sizes of the intermediate result increase, which leads to longer execution times in Fig. 4a. This results from the large FPRs corresponding to the increased number of join keys that are inserted into the filters. In TMFR-Join, as shown in Fig. 5b, the intermediate result size increases when σ_L is greater than or equal to 0.5. As σ_L is larger than the given FPR threshold, the filters are disabled and the join is executed like the repartition join. Although the size of the intermediate results increases, TMFR-Join exhibits better performance than the existing MFR-Join because the costs of creating, merging, and checking the filters are saved.

Figures 6 and 7 show the execution times of TMFR-Join with Bloom filters and Interval filters, respectively, with varying FPR thresholds. It can be seen that filters are advantageous when σ_L is 0.3, while they are disadvantageous when σ_L is 0.5 or 0.8.

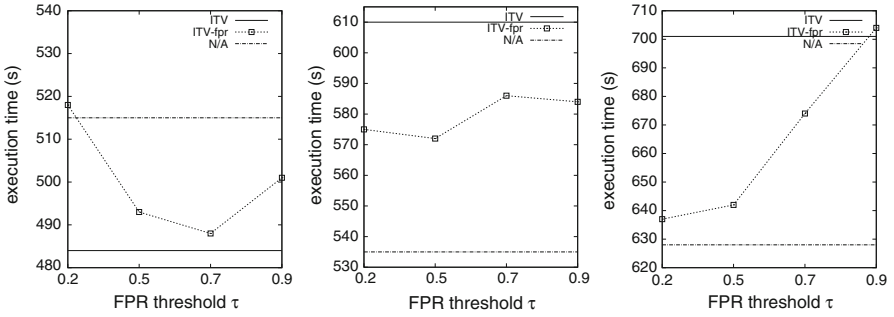

Fig. 7 Execution times of TMFR-Join with interval filters varying FPR thresholds. **a** $\sigma_L = 0.3$, **b** $\sigma_L = 0.5$, **c** $\sigma_L = 0.8$

If the FPR threshold is too small, as with τ of 0.2 in Figs. 6a and 7a, the filters may be disabled despite being beneficial to performance. Conversely, if the FPR threshold is too large, as with τ of 0.9 in Figs. 6c and 7c, the filters may not be disabled despite being detrimental to performance. Although it cannot be seen clearly due to the experimental error in Figs. 6b and 7b, low FPR threshold values lead to better performance by disabling filters. Consequently, it is important to set the appropriate FPR threshold for each query, as stated in Sect. 4.3, so that it exhibits performance that is close to that exhibited by the better join processing techniques.

6 Conclusions and future work

In this paper, we presented a join processing method with threshold-based filtering, called TMFR-Join. TMFR-Join estimates the performance of filters by means of FPRs, and disables those filters whose FPRs are greater than a user-configured threshold. It also enables the application of various filtering techniques so long as they support certain operations. The experimental results show that our techniques have stable performance close to that of the better of the existing join algorithms, both with and without filters. For future work, we plan to integrate our framework with large-scale data warehouse systems or join-based applications like a graph pattern matching framework.

Acknowledgments This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIP) (No. 20120005695), by ETRI R&D Program ("Development of Big Data Platform for Dual Mode Batch/Query Analytics, 14ZS1400") funded by the Government of Korea, and by Samsung Electronics Co. Ltd.

References

1. Thusoo A, Antony S, Jain N, Murthy R, Shao Z, Borthakur D, Sarma JS, Liu H (2010) Data warehousing and analytics infrastructure at facebook. In: Proceedings of the 2010 ACM SIGMOD international conference on management of data, SIGMOD'10, pp 1013–1020

2. Gupta R, Gupta H, Nambiar U, Mohania M (2010) Efficiently querying archived data using hadoop. In: Proceedings of the 19th ACM international conference on information and knowledge management, CIKM'10, pp 1301–1304
3. Dean J, Ghemawat S (2004) Mapreduce: simplified data processing on large clusters. In: Proceedings of the 6th USENIX symposium on opearting systems design and implementation, OSDI'04, pp 137–150
4. Hadoop. http://hadoop.apache.org/. Accessed 3 April 2014
5. Blanas S, Patel JM, Ercegovac V, Rao J, Shekita EJ, Tian Y (2010) A comparison of join algorithms for log processing in mapreduce. In: Proceedings of the 2010 ACM SIGMOD international conference on management of data, SIGMOD'10, pp 975–986
6. Yang HC, Dasdan A, Hsiao RL, Parker DS (2007) 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, pp 1029–1040
7. Espinosa A, Hernandez P, Moure JC, Protasio J, Ripoll A (2012) Analysis and improvement of map-reduce data distribution in read mapping applications. J Supercomput 62(3):1305–1317
8. Bloom BH (1970) Space/time trade-offs in hash coding with allowable errors. Commun ACM 13(7):422–426
9. Koutris P (2011) Bloom filters in distributed query execution. University of Washington, Washington. http://www.cs.washington.edu/education/courses/cse544/11wi/projects/koutris.pdf. Accessed 3 April 2014
10. Lee T, Kim K, Kim HJ (2012) Join processing using bloom filter in mapreduce. In: Proceedings of the 2012 ACM research in applied computation symposium, RACS'12, pp 100–105
11. Palla K (2009) A comparative analysis of join algorithms using the hadoop map/reduce framework. Master's thesis, University of Edinburgh, Edinburgh
12. Zhang C, Wu L, Li J (2013) Efficient processing distributed joins with bloomfilter using mapreduce. Int J Grid Distrib Comput 6(3):43–58
13. Tarkoma S, Rothenberg CE, Lagerspetz E (2012) Theory and practice of bloom filters for distributed systems. IEEE Commun Surv Tutor 14(1):131–155
14. Bender MA, Farach-Colton M, Johnson R, Kraner R, Kuszmaul BC, Medjedovic D, Montes P, Shetty P, Spillane RP, Zadok E (2012) Don't thrash: how to cache your hash on flash. Proc VLDB Endow 5(11):1627–1637
15. Quislant R, Gutierrez E, Plata O, Zapata EL (2010), Interval filter: a locality-aware alternative to bloom filters for hardware membership queries by interval classification. In: Proceedings of the 11th international conference on intelligent data engineering and automated learning, IDEAL'10, pp 162–169
16. Lee KH, Lee YJ, Choi H, Chung YD, Moon B (2011) Parallel data processing with mapreduce: a survey. ACM SIGMOD Rec 40(4):11–20
17. White T (2011) Hadoop: the definitive guide, 2nd edn. O'Reilly Media Inc., USA
18. Afrati FN, Ullman JD (2010) Optimizing joins in a map-reduce environment. In: Proceedings of the 13th international conference on extending database technology, EDBT'10, pp 99–110
19. Jiang D, Tung AKH, Chen G (2011) Map-join-reduce: toward scalable and efficient data analysis on large clusters. IEEE Trans Knowl Data Eng 23(9):1299–1311
20. Mackert LF, Lohman GM (1986) R* optimizer validation and performance evaluation for distributed queries. In: Proceedings of the 12th international conference on very large data bases, VLDB'86, pp 149–159
21. Kemper A, Kossmann D, Wiesner C (1999), Generalized hash teams for join and group-by. In: Proceedings of the 25th international conference on very large data bases, VLDB'99, pp 30–41
22. Michael L, Nejdl W, Papapetrou O, Siberski W (2007) Improving distributed join efficiency with extended bloom filter operations. In: Proceedings of the 21st international conference on advanced networking and applications, AINA'07, pp 187–194
23. Ramesh S, Papapetrou O, Siberski W (2008) Optimizing distributed joins with bloom filters. Distributed computing and internet technology. In: Lecture notes in computer science, vol 5375, pp. 145–156
24. Papapetrou O, Siberski W, Nejdl W (2010) Cardinality estimation and dynamic length adaptation for bloom filters. Distrib Parallel Databases 28(2–3):119–156
25. Herodotou H (2011) Hadoop performance models. In: Technical report CS-2011-05, Duke University, Durham. http://www.cs.duke.edu/starfish/files/hadoop-models.pdf. Accessed 3 April 2014
26. Cluster setup. http://hadoop.apache.org/docs/r0.19.1/cluster_setup.html. Accessed 3 April 2014
27. TPC-H benchmark. http://www.tpc.org/tpch/. Accessed 3 April 2014