Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2014, Article ID 249418, 11 pages
http://dx.doi.org/10.1155/2014/249418

Research Article

Application of Filters to Multiway Joins in MapReduce

Taewhi Lee,[1] Dong-Hyuk Im,[2] Hangkyu Kim,[3] and Hyoung-Joo Kim[1]

[1] *School of Computer Science and Engineering, Seoul National University, Seoul 151-744, Republic of Korea*
[2] *Department of Computer and Information Engineering, Hoseo University, Asan, Chungnam 336-795, Republic of Korea*
[3] *Software Center, Samsung Electronics, Suwon, Gyeonggi 443-370, Republic of Korea*

Correspondence should be addressed to Dong-Hyuk Im; dhim@hoseo.edu

Received 6 November 2013; Revised 23 December 2013; Accepted 30 December 2013; Published 4 March 2014

Academic Editor: Cheng Shao

Joining multiple datasets in MapReduce may amplify the disk and network overheads because intermediate join results have to be written to the underlying distributed file system, or map output records have to be replicated multiple times. This paper proposes a method for applying filters based on the processing order of input datasets, which is appropriate for the two types of multiway joins: common attribute joins and distinct attribute joins. The number of redundant records filtered depends on the processing order. In common attribute joins, the input records do not need to be replicated, so a set of filters is created, which are applied in turn. In distinct attribute joins, the input records have to be replicated, so multiple sets of filters need to be created, which depend on the number of join attributes. The experimental results showed that our approach outperformed a cascade of two-way joins and basic multiway joins in cases where small portions of input datasets were joined.

1. Introduction

Join processing in MapReduce [1] has attracted the attention of researchers in recent years. This is because MapReduce does not support join operations directly, although it is a useful framework for large-scale data analysis. In particular, joining multiple datasets in MapReduce has been a challenging problem because it may amplify the disk and network overheads. Multiple datasets can be joined in the following two ways: (1) using a cascade of two-way (or smaller multiway) joins and (2) with a single multiway join. However, both methods have some drawbacks. A cascade of two-way joins has to write the intermediate join results to the underlying distributed file system, which generally replicates multiple records to ensure high availability and fault tolerance. To process multiway joins in a single MapReduce job, the map output records have to be replicated multiple times, instead of writing only the final join results to the distributed file system.

Previous studies have attempted to improve join performance using filtering techniques [2–5], including our previous study [6, 7]. These studies focused on reducing the number of map output records that are not joined. This may be more beneficial with multiway joins. The map output records are replicated multiple times, so filtering out redundant records removes multiple copies of the record in multiway joins. Figure 1 shows an example of basic multiway join processing in MapReduce. In this example, three input datasets, that is, R(a,b), S(b,c), and T(c,d), are joined with two attributes b and c. To join the three datasets simultaneously, some datasets need to be replicated, that is, R and T in this example. Replication may degrade the join performance, so it is important to reduce the number of redundant records, which are marked with strikethroughs in Figure 1.

In this study, we extend the concept of filtering techniques to multiway joins. Multiway joins can be classified into two types: *common attribute joins* and *distinct attribute joins* [8]. A common attribute join combines datasets based on one or more shared attributes, whereas some relations do not have join attributes in a distinct attribute join. The example shown in Figure 1 illustrates a distinct attribute join, because R does not have the attribute c and T does not have the attribute b. Thus, we propose methods for filter application that are suitable for both cases. We do not limit the filtering technique used and various techniques can be applied. Simple hash filters were used for the evaluation. The contributions of this study are as follows.

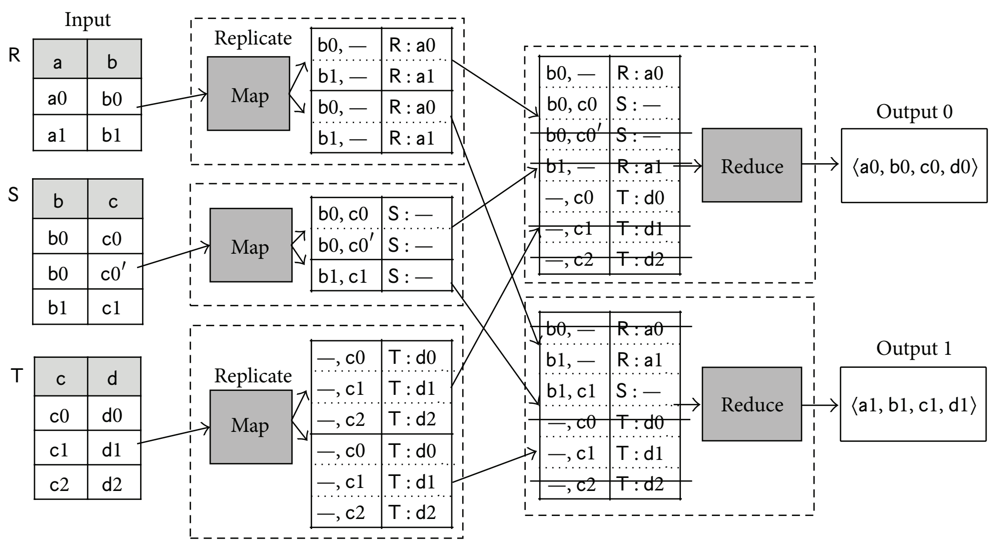


Figure 1: Basic multiway join processing in MapReduce.

(i) We propose methods for applying filters to multiway joins. For common attribute joins, a set of filters is created and applied in turn. For distinct attribute joins, multiple sets of filters are created, according to the number of join attributes. The filters can be applied in various patterns, according to the processing order of the input datasets.

(ii) We provide specific details of our prototype implementation. We briefly introduce MFR-Join, which is our general join framework with filtering techniques implemented in Hadoop [9]. Based on this framework, algorithms that perform multiway joins with filters are described.

(iii) Experimental results obtained using our basic framework are presented. Our proposed techniques were evaluated based on comparison with a cascade of two-way joins and a multiway join without filters. The results showed that our approach outperformed the other techniques in cases where small portions of the input datasets were joined.

The remainder of this paper is organized as follows. Section 2 reviews background information and previous research related to this study. Section 3 explains our proposed methods for applying filters to multi-way joins. Section 4 describes the implementation details of our methods. Section 5 discusses our experimental results. Finally, we conclude this paper in Section 6.

2. Background and Related Work

This section presents the basic concepts and previous research related to our study. Section 2.1 explains the basis framework, MapReduce. Next, two-way and multiway join algorithms in MapReduce are reviewed in Sections 2.2 and 2.3.

2.1. MapReduce. MapReduce [1] is Google's programming model for large-scale data processing, which is run on a shared-nothing cluster. MapReduce liberates users from the responsibility of implementing parallel and distributed processing features by providing them automatically. Thus, users only have to write MapReduce programs with two functions: *map* and *reduce*. The map function takes a simple key/value pair as its input and it produces a set of intermediate key/value pairs. The reduce function takes an intermediate key and a set of values that correspond to the key as its input, and it generates the final output key/value pairs.

A MapReduce cluster comprises one master node and a number of worker nodes. When a MapReduce job is submitted, the master node creates the map, reduces tasks, and assigns each task to idle workers. A map worker reads the input split and executes the map function submitted by the user. The map output records are grouped and sorted by the key and then stored in partitions for each reduce worker. A reduce worker reads its corresponding partitions from all the map workers, merges the partitions, and executes the reduce function. When all of the tasks are complete, the MapReduce job is finished.

Hadoop [9] is a popular open-source implementation of the MapReduce framework. In Hadoop, the master node is called the *jobtracker* and the worker node is called the *tasktracker*. Tasktrackers run one or more *mapper* and *reducer* processes, which execute map and reduce tasks, respectively. The proposed method was implemented using Hadoop, so the Hadoop terminology is used in the remainder of this paper.

2.2. Join Processing in MapReduce. Join algorithms in MapReduce are classified roughly into two categories: map-side joins and reduce-side joins [10]. Map-side joins produce the final join results in the map phase and do not use the reduce phase. They do not need to pass intermediate results from mappers to reducers, which means that map-side joins are more efficient than reduce-side joins, although they can only be used in particular circumstances. Hadoop's map-side join [11], referred to as the map-merge join [10], merges input datasets that are partitioned and sorted on the join keys in the same manner, which is similar to the merge join in traditional DBMS. An additional MapReduce job is required if the input datasets are not partitioned and sorted in advance. The broadcast join [12] distributes the smaller of the input datasets to all of the mappers and performs the join in the map phase. This approach is efficient only if the input dataset is small.

Reduce-side joins can be used in more general cases, but they are inefficient because large intermediate records are sent from mappers to reducers. The repartition join [12] is the most common join algorithm in MapReduce, but all of the input records have to be sent to reducers, including redundant records that are not relevant to the join. This may lead to a performance bottleneck. The semijoin in MapReduce [12] works in a similar manner to semijoin in traditional DBMS. This approach may reduce the size of the intermediate results by filtering out the unreferenced records with unique join keys. Therefore, it is efficient when small portions of records participate in joins. However, the semijoin requires three MapReduce jobs, which means that the results of each job are written and read in the next job. This incurs additional I/O overheads.

Recent studies have attempted to adapt the bloomjoin [13], which filters out tuples that do not participate in a join using Bloom filters [14], to the MapReduce framework. Reduce-side joins with a Bloom filter were proposed previously [2, 4, 5], but they create the filter via an independent job. Therefore, they have to process the input datasets multiple times. Koutris [3] theoretically investigated join methods using Bloom filters within a single MapReduce job but did not provide specific technical details.

2.3. Multiway Joins in MapReduce. Several datasets can be joined simultaneously in a single MapReduce job by replicating some input datasets, as shown in Figure 1. Previous studies have also attempted to optimize the number of input records replicated in multiway joins [15, 16] and they use similar methods for minimizing the number. Figure 2 shows a partial replication of the input records for a join example with three datasets and nine reducers. Unlike the naive multiway join that replicates some input datasets fully, the input records of R and T are replicated by only three reducers, depending on the hash values of the join attributes b and c. This optimization problem can be formulated as a problem of minimizing the total number of records that are sent to reducers. Afrati and Ullman [15] solved the minimization problem using a method based on Lagrangian multipliers to find the optimal solution. Jiang et al. [16] used a heuristic approach to find an approximate solution. These studies can be used in combination with our approach, which uses filtering techniques to facilitate more efficient multi-way join processing.

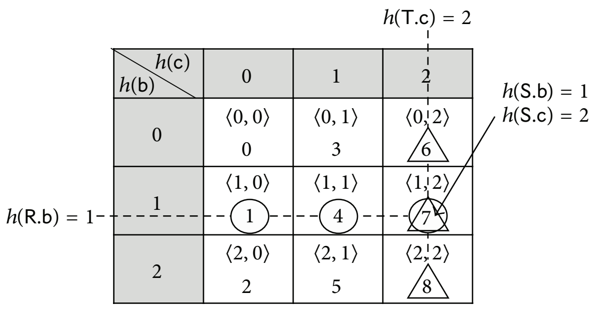


Figure 2: Optimizing multiway joins in MapReduce.

3. Application of Filters to Multiway Joins

This section presents the basic methods used to create and apply filters to common and distinct multiway joins. To simplify the discussion, joins between three datasets are considered in the following subsections. We then consider the processing of general joins between multiple datasets.

3.1. Common Attribute Joins. In common attribute joins, all of the input datasets share join attributes. In these cases, the input records do not need to be replicated and they can be processed in a similar manner to two-way joins. A set of filters is created and probed in turn, depending on the processing order of the input datasets.

Figure 3 shows an example of a common attribute join between three input datasets, that is, R(a,b), S(a,c), and T(a,d), based on the attribute a. Similar to two-way joins, the input records of the first dataset, that is, R in the figure, are not filtered out and they are used to create a set of filters for the next dataset. The input records of the second dataset, that is, S in the figure, are processed using the filters and some redundant records can be filtered out. In addition, another set of filters is created using the map output records from S for the next dataset. The map output records are contained in the first set of filters, which means that the second set of filters is automatically the same as the intersection of the filters that are created independently using the first and second dataset. Finally, the third input dataset, that is, T in the figure, is processed with the second set of filters. Another set of filters does not need to be created because this is the final input dataset for the join attribute.

The input datasets are processed in the order of R, S, and T in this example, but any order can be processed in the same way. The join cost depends on the number of input records, the ratio of the joined records, and the false positive rate of the filters.

3.2. Distinct Attribute Joins. In distinct attribute joins, the input datasets may not have some join attributes. Thus, some of the datasets with missing attributes need to be replicated because their records may be joined to the input records of other datasets with any values of the missing attributes. Let us consider the join example shown in Figure 1, which is a join between three input datasets, that is, R(a,b), S(b,c), and T(c,d), based on two attributes, that is, b and c. We assume

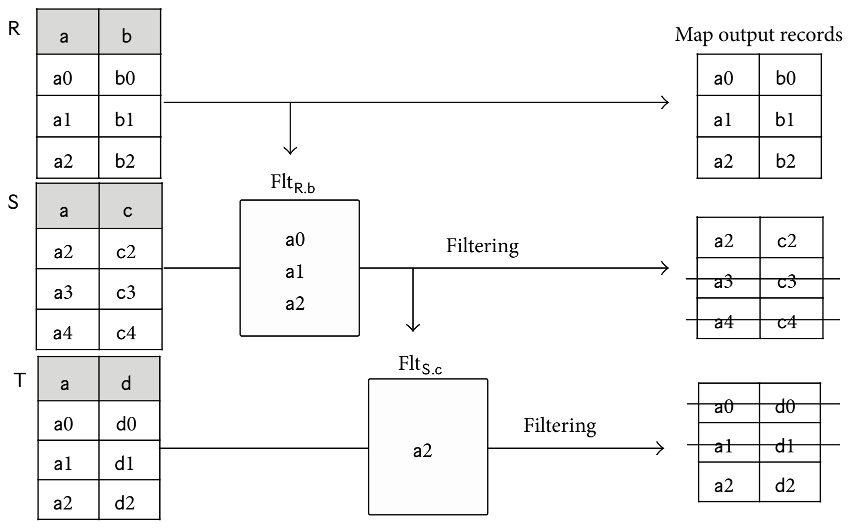


FIGURE 3: Common attribute join.

that R and T are replicated by two reducers in this example. The example join can be processed in 3! = 6 different orders of the input datasets. Depending on the processing order, the filters can be applied in three patterns: *chain*, *star-fact*, and *star-dim*.

3.2.1. Chain. The chain pattern creates and probes filters in turn, in a similar manner to common attribute joins, except that each set of filters is created for a different join attribute. This is analogous to the indirect partitioning method proposed by Kemper et al. [17]. Two processing orders correspond to this pattern, that is, R-S-T and T-S-R. Figure 4(a) illustrates an example of a distinct attribute join with the chain pattern. The first dataset R is replicated by reducers and a set of filters is created with the values of the join attribute, b in R. The second dataset S is processed using the filters and some redundant records may be filtered out. Meanwhile, another set of filters is created using the map output records from S based on the other join attribute c. Next, the third dataset in the figure, T, is replicated and processed with the second set of filters.

3.2.2. Star-Fact. The star-fact pattern creates filters using the dataset with both join attributes and uses the filters to process the other datasets. In database terms, a fact table in a star join is used to create the filters. Two processing orders correspond to this pattern, that is, S-R-T and S-T-R. Figure 4(b) shows an example of a distinct attribute join with the star-fact pattern. The first dataset S, which has both join attributes, is processed and two sets of filters for each join attribute, that is, b and c, are created. Next, the other datasets, that is, R and T, are replicated and processed using the set of filters that correspond to the join attribute.

3.2.3. Star-Dim. The star-dim pattern creates filters using the datasets with missing join attributes and uses the filters to process the other dataset. In database terms, the dimension tables in a star join are used to create the filters. The remaining two cases, that is, R-T-S and T-R-S, correspond to this pattern. Figure 4(c) shows an example of a distinct attribute join with the star-dim pattern. The first and second datasets, that is, R and T, are replicated and processed without filters. Each set of filters for the join attributes is created using their join attribute values. Next, the third dataset S is processed using both filters and some redundant records are filtered out. The star-dim pattern appears to be inefficient in this example, but it is efficient if the number of records in the third dataset is much larger than those in the other datasets.

3.3. General Multiway Joins. The basic filtering patterns are presented in Sections 3.1 and 3.2. Multiway joins of more than three datasets can be processed by combining these patterns. In general, multiple datasets can be joined in any processing order for input datasets using the following rules.

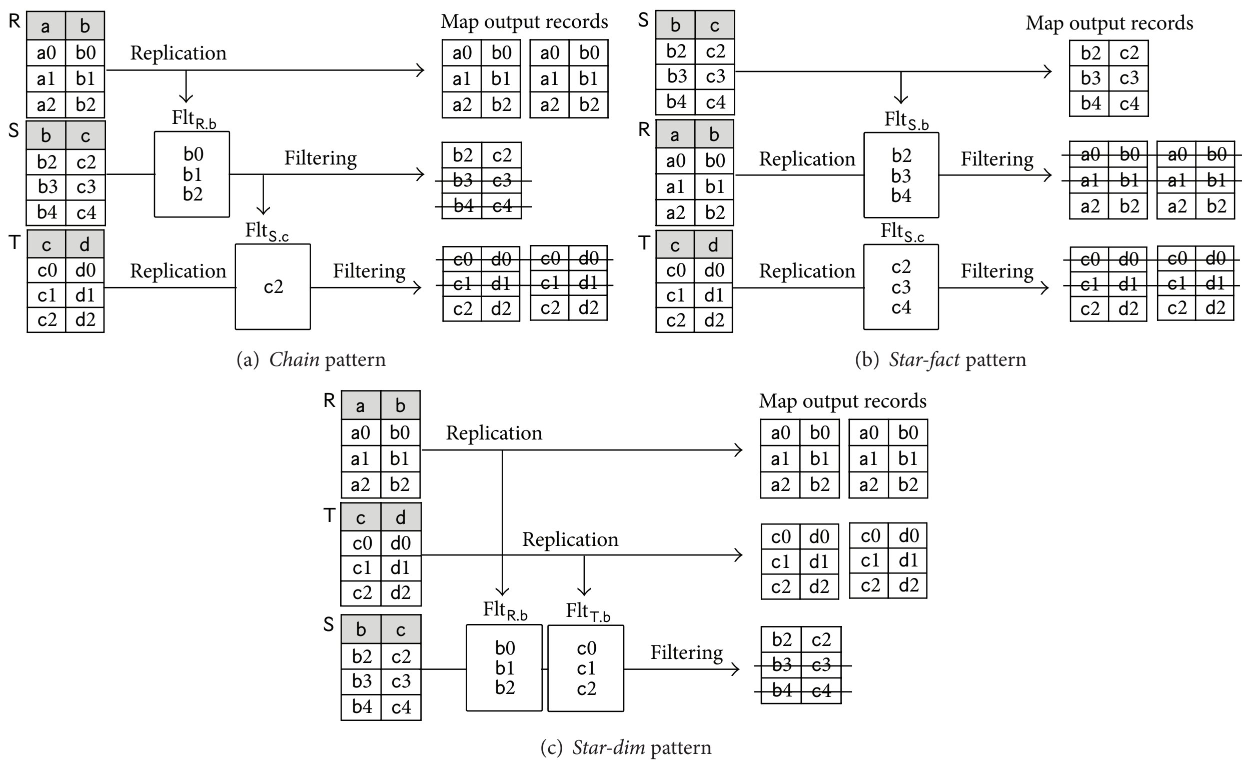
For each join attribute,

(i) create a set of filters if the dataset is not the last one with the attribute;

(ii) probe the existing set of filters if the dataset is not the first one with the attribute.

All combinations of these patterns can be summarized using these rules. The filters can be applied in any processing order, but the processing order must be selected carefully because it affects the join cost.

3.4. Cost Analysis. The number of intermediate map output records is the most important factor that influences the overall cost. For multi-way joins, the number is affected by the replication factors for each join attribute. Let n_i be the number of records in the ith input dataset and let f_i be the replication factor for the ith join key. Next, let σ_{i_j} be the ratio of joined records in the ith dataset relative to a previously processed jth dataset, and let p_{i_j} be the false positive probability of the previously created filters for the jth join attributes when the



(a) *Chain* pattern (b) *Star-fact* pattern (c) *Star-dim* pattern

Figure 4: Distinct attribute joins.

ith dataset is processed. In addition, let $c(i, j)$ be a binary value function that returns whether the ith dataset contains the jth join attribute as follows:

$$c(i,j) = \begin{cases} 1, & \text{if } i\text{th dataset contains the } j\text{th join attribute,} \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

Assuming that the attribute values of the input datasets are independent, the number of intermediate map output records n^f_{inter} in a multi-way join between n datasets based on k join attributes can be expressed as follows:

$$n^f_{\text{inter}} = \sum_{i=1}^{k} \left(f_i \cdot n_i \cdot S_i + f_i \cdot n_i \cdot (1 - S_i) \cdot \prod_{j=1}^{k} \left(p_{i\_j} \cdot c(i,j) \right) \right), \tag{2}$$

where

$$S_i = \begin{cases} 1, & \text{if } i = 1, \\ \prod_{j=1}^{i-1} \sigma_{i\_j}, & \text{otherwise,} \end{cases} \tag{3}$$

$$\prod_{i=1}^{k} f_i = (\text{\# of reducers}).$$

We need to find the replication factors and processing order for input datasets that minimizes the number of intermediate map output records. Note that the factors f_i, S_i, and p_{i_j} depend on the processing order of the input datasets. These equations can be used to select the processing order and to estimate the join cost, but there may be a large search space if the numbers of reducers and the input datasets are large. In these cases, the factors have to be selected using heuristics. For example, the replication factors can be computed using the method proposed by Afrati and Ullman [15], which does not consider filters, or they can be predefined by users or determined by optimizer modules. The brute force approach was used in the experiments conducted with ten reducers in the present study.

4. Implementation Details

In this section, we briefly introduce our basic framework, which is called MFR-Join. The two major issues when processing multi-way joins simultaneously in the framework are replicating the records for the corresponding reducers and processing multiple join attributes for filtering. The following subsections describe the specific implementation details that address these issues.

4.1. MFR-Join. MFR-Join is a general join framework with filtering techniques in MapReduce, which is proposed in our previous studies [6, 7]. MFR-Join, which was implemented in Hadoop, has two major differences from the original Hadoop

```
    ▷ $n$: the number of join attributes
    ▷ $joinAttr$: an array of join attribute values (some values may be missing)
    ▷ $repl$: an array of replication factors for each join attribute
(1) procedure FindTargetReducers($joinAttr[1, \dots, n]$, $repl[1, \dots, n]$)
(2)     $reducerList \leftarrow \emptyset$
(3)     $coord \leftarrow \emptyset$
(4)     for $i = 1$ to $n$ do
(5)         if $joinAttr[i]$ is not null then
    ▷ $GetPartition()$: returns a number in the range [0 and ($repl\ [i] - 1$)]
(6)             $coord[i] \leftarrow GetPartition(joinAttr[i], repl[i])$
(7)         else
(8)             $coord[i] \leftarrow 0$
(9)         end if
(10)    end for
(11)    while $true$ do
(12)        $rid \leftarrow CoordToReducer(coord, repl)$
(13)        $reducerList \leftarrow reducerList \cup rid$
(14)        if $IncrCoord(coord, joinAttr, repl) = false$ then
(15)            break
(16)        end if
(17)    end while
(18)    return $reducerList$
(19) end procedure
```

Algorithm 1: Finding target reducers.

```
(1) procedure COORDTOREDUCER($coord[1, \dots, n], repl[1, \dots, n]$)
(2)     $reducer \leftarrow 0$
(3)     $base \leftarrow 1$
(4)     for $i = 1$ to $n$ do
(5)         $reducer \leftarrow reducer + base * coord[i]$
(6)         $base \leftarrow base * repl[i]$
(7)     end for
(8)     return $reducer$
(9) end procedure
```

Algorithm 2: Converting a coordinate to a reducer id.

```
(1) procedure INCRCOORD($coord[1, \dots, n]$, $joinAttr[1, \dots, n]$, $repl[1, \dots, n]$)
(2)     for $i = 1$ to $n$ do
(3)         if $joinAttr[i]$ is not null then
(4)             continue
(5)         end if
(6)         $coord[i] \leftarrow coord[i] + 1$
(7)         if $coord[i] < repl[i]$ then
(8)             return $true$
(9)         end if
(10)        $coord[i] \leftarrow 0$
(11)    end for
(12)    return $false$
(13) end procedure
```

Algorithm 3: Increasing a coordinate.

```
    ▷ value: an input record from the ith dataset
    ▷ repl: replication factors that are pre-computed or pre-defined in the init phase
(1) procedure MAP($key$, $value$)
(2)     extract the join attribute values $joinAttr[1, \ldots, n]$ by parsing the input record value
                                        ▷ ‖: concatenation, #: delimiter
(3)     $joinAttrKey \leftarrow joinAttr[1] \| \# \| \cdots \| \# \| joinAttr[n]$
(4)     $reducerList \leftarrow FindTargetReducers(joinAttr, repl)$
(5)     for each reducer in reducerList do
                                        ▷ tag: the dataset id of the record
(6)         $Emit((joinAttrKey, reducer), (value, tag))$
(7)     end for
(8) end procedure
```

Algorithm 4: Map function.

```
    ▷ values: intermediate records (record, datasetId) with the same reducer id
(1) procedure REDUCE($key$, $values$)
    ▷ recordList: lists for buffering the intermediate records based on their dataset id
(2)     $recordList \leftarrow \emptyset$
(3)     for each value in values do
(4)         $dsid \leftarrow value.tag$
(5)         $recordList[dsid] \leftarrow recordList[dsid] \cup value.record$
(6)     end for
    ▷ Join(recordList): returns the join results between the records in each record list
(7)         $Emit\ (Join\ (recordList))$
(8) end procedure
```

Algorithm 5: Reduce function.

system. First, the map tasks are scheduled according to the order of the dataset, whereas Hadoop assigns map tasks based on the order of the input split size. This allows us to apply database techniques such as tuple filtering and join ordering. Second, the filters are constructed dynamically within a single MapReduce job. The filters are created based on the first input dataset in a distributed manner to filter out the second input dataset. In this way, MFR-Join can reduce the communication cost for redundant records by processing the input datasets only once. Our previous studies [6, 7] provide further details of these processes.

4.2. Partition Assignment. For each input dataset, its corresponding reducers for replication are determined as shown in Figure 2. The replication of input records for their corresponding reducers can be implemented in a similar manner to the data partitioning method described by Zhang et al. [18]. Algorithm 1 demonstrates how to find the target reducers that correspond to an input record. Depending on the number of join attributes n, we may assume that there is an n-dimensional space with integer coordinates, where each dimension represents each join attribute and a position represents a reducer. Note that n is the number of join attributes, rather than the number of input datasets, which was the case in a previous study [18] that aimed to process theta joins. Then, the corresponding positions of an input record can be obtained by partitioning the join attribute values of the record in a range from zero to the corresponding replication factor-1. A coordinate for a missing join attribute can be expressed using a special character, that is, "$*$." This indicates that the record corresponds to the reducers with all possible values for the coordinate. Next, the positions are converted into integer identifiers of the reducer by adding up the values of each position, which are multiplied by the replication factors for the preceding dimensions. Algorithms 2 and 3 show the pseudocodes for the conversion process.

Now, we consider an example of three-way joins between R(a,b), S(b,c), and T(c,d) using the two join attributes shown in Figure 2. We assume that the number of reducers is nine and that there are three replication factors for both R and T. The positions that correspond to each record of R, S, and T in the figure are $\langle 1, * \rangle$, $\langle 1, 2 \rangle$, and $\langle *, 2 \rangle$, respectively. $\langle 1, * \rangle$ represents the positions $\langle 1, 0 \rangle$, $\langle 1, 1 \rangle$, and $\langle 1, 2 \rangle$. If we assign each reducer with an integer identifier from zero to eight incrementally, starting from the top-left cell in a vertical direction, the identifiers of the reducers that correspond to the records for R are 1, 4, and 7. Similarly, because $\langle 2 \rangle$ represents the positions $\langle 0, 2 \rangle$, $\langle 1, 2 \rangle$, and $\langle 2, 2 \rangle$, the identifiers of the reducers that correspond to the records of T are 6, 7, and 8. The position of the record of S is $\langle 1, 2 \rangle$, so the identifier of its corresponding reducers is 7. Thus, these records are gathered and joined by the reducer with the identifier 7.

4.3. MapReduce Functions. A prototype MFR-Join framework has been implemented to create and probe filters using the keys of map output records. To process distinct attribute multi-way joins with multiple join attributes, the keys need to be separated with a delimiter, which is configured using the additional parameter `mapred.filter.key.delimiter`. The target reducers for a record can be found using Algorithm 1, which is described in Section 4.2. Algorithm 4 is the pseudocode for the map function used in multiway joins.

The records generated by the map function are then processed by the MFR-Join framework, as explained in Section 3. Some redundant records will be filtered out, depending on the processing order of the input datasets. The map output records that passed the filters have been gathered in the corresponding reducers by their reducer identifiers. Using the reduce function, the records are classified based on the tag representing their original dataset and they are joined with traditional join algorithms. Algorithm 5 is the pseudo-code for the reduce function in multi-way joins.

5. Performance Evaluation

In this section, we present our experimental results for common and distinct attribute joins. All of the experiments were run on a cluster of 11 machines, which comprised one jobtracker and 10 tasktrackers. Each machine had 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.

Our proposed framework was implemented in Hadoop 0.20.2. The Hadoop distributed file system (HDFS) was set to use 128 MB blocks and to replicate them three times. Each tasktracker could run three map tasks and one reduce task simultaneously. The I/O buffer was set to 128 KB and the memory used to sort the data was set to 200 MB.

5.1. Common Attribute Joins. We used TPC-H benchmark [19] datasets with a scale factor of 100 for evaluation. The scale factor was the size of the entire database in gigabytes. We performed a join between three tables in the database, that is, `part`, `partsupp`, and `lineitem`, which had a common attribute, that is, partkey. The sizes of the datasets are shown in Table 1.

Our test query was extracted from TPC-H Q9 and it could be expressed in SQL-like syntax as follows:

```
SELECT l.orderkey, l.partkey,
       l.suppkey, ps.suppkey,
       l.extendedprice * (1 − l.discount)
       − ps.supplycost * l.quantity as profit
FROM part p, partsupp ps, lineitem l
WHERE p.partkey = l.partkey
  AND ps.partkey = l.partkey
  AND p.name like '%green'
  AND ps.supplycost < c_ps.
```

To control the amount of joined records, the selection predicate `ps.supplycost` $< c_{ps}$ was added to the query. The attribute `ps.supplycost` had a decimal value in the range of 1.0 to 1000.0. We ran the query and changed the predicate value c_{ps} in the predicate with increments of one hundred. Thus, the ratio of records that satisfied the predicate in `partsupp` σ_{ps} was changed by about 10%. We compared the performance of our join method with a repartition join without filters, because common attribute joins can be processed without replication in a single MapReduce job. In our method, simple hash filters with a size of 8 Mb were used and the input datasets were processed in the following order: `part`, `partsupp`, and `lineitem`.

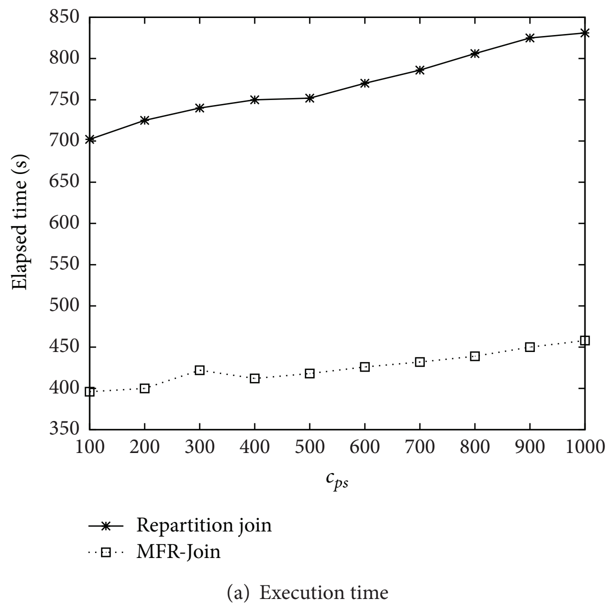
TABLE 1: Test datasets for common attribute joins.

Table	Number of records	Size	Number of records satisfying selection predicate
Part	20 M	2.3 GB	1.08 M
Partsupp	80 M	12 GB	80 M $* \sigma_{ps}$
Lineitem	600 M	75 GB	600 M (no predicate)

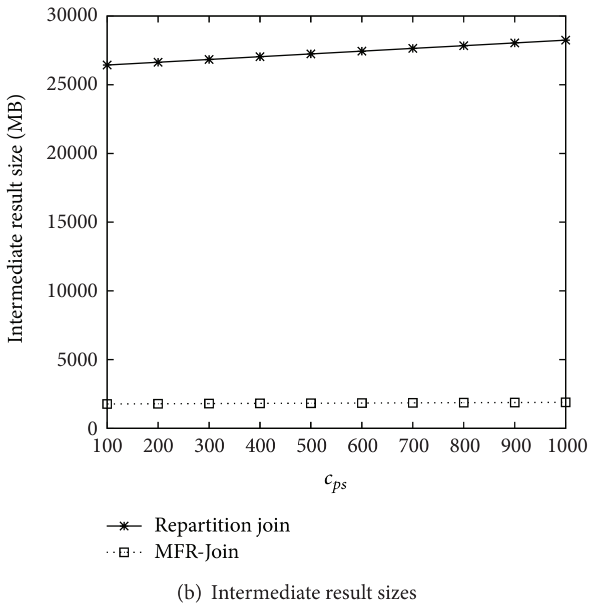
We compared the performance of our techniques to that of the existing repartition join [12] because input records do not need to be replicated. Although there are a few existing techniques such as the semijoin [12] with Bloom filters [2], we did not compare them as our previous paper showed that MFR-Join outperforms them [7], and they do not support multiway joins directly. Figure 5 shows the execution times and the sizes of the intermediate results for the test queries. The results showed that our method significantly outperformed the repartition join for all of the test cases in Figure 5(a). This is because large numbers of redundant intermediate results from the `lineitem` dataset are filtered out using our method, as shown in Figure 5(b). The `lineitem` dataset has no selection predicate, so the repartition join has to generate the entire dataset as intermediate results.

5.2. Distinct Attribute Joins. For distinct attribute joins, the TPC-H benchmark [19] datasets were also used, but with a scale factor of 300. We performed the following join query, which was extracted from TPC-H Q2, between the following five tables: `nation`, `region`, `supplier`, `part`, and `partsupp`, where the sizes are as shown in Table 2. The two tables, `nation` and `region`, contained just a few records, so we treated the joins of the tables as in-memory hashing and excluded the two tables from Table 2.

```
SELECT s.acctbal, s.name, n.name,
       p.partkey, ps.supplycost, p.mfgr,
       s.address, s.phone, s.comment
FROM nation n, region r, supplier s,
     part p, partsupp ps
WHERE n.regionkey = r.regionkey
  AND r.name = 'EUROPE'
  AND s.nationkey = n.nationkey
  AND s.suppkey = ps.suppkey
  AND p.partkey = ps.partkey
  AND p.type like '%BRASS'
  AND p.size <= c_p.
```



(a) Execution time



(b) Intermediate result sizes

Figure 5: Performance of common attribute joins.

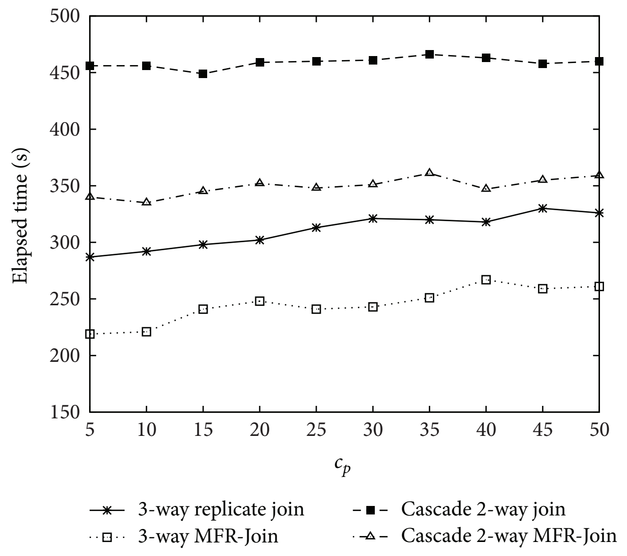
Table 2: Test datasets for distinct attribute joins.

Table	Number of records	Size	Number of records satisfying selection predicate
Supplier	3 M	0.4 GB	0.6 M
Part	60 M	6.9 GB	60 M $*$ 0.2 $* \sigma_p$
Partsupp	240 M	35 GB	240 M (no predicate)

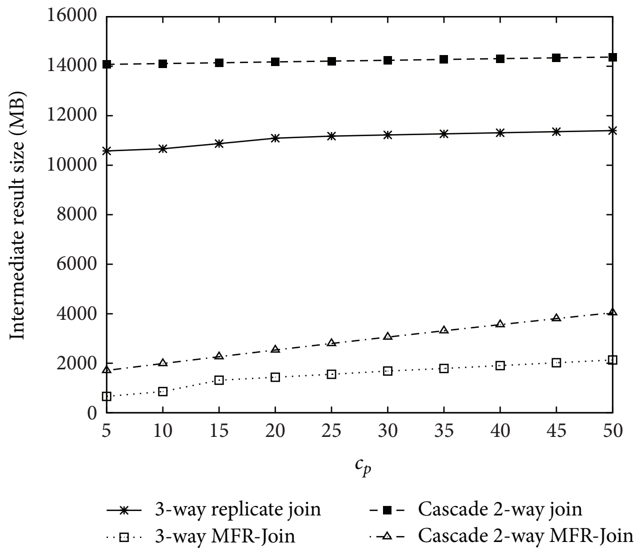
Similar to the test query for common attribute joins, we added a selection predicate, `p.size` <= c_p, to control the amount of joined records. The attribute `p.size` had an integer value in the range of 1 to 50 and we ran the query by changing the predicate value c_p with increments of five. Thus, the ratio of records that satisfied the predicate in `part` σ_p was changed by about 10%. Distinct attribute joins required the replication of some input datasets, so we selected the best from the results using all possible combinations of the replication factors.

We compared the performance of our multi-way join method (denoted as 3-way MFR-Join) with that of the basic multi-way join (denoted by 3-way replicate join) and with that of the cascade of two-way joins with and without filters (denoted by Cascade 2-way MFR-Join and Cascade 2-way join, resp.). Simple hash filters with a size of 8 Mb were also used for the multi-way join and the cascade of two-way MFR-Join. In the two-way joins, `supplier` and `partsupp` were joined first, before the intermediate join results and `part` were joined. Figure 6 shows the execution times and intermediate result sizes for the test queries. The results of our three-way MFR-Join with the star-dim filtering pattern had the best performance with the queries. The multi-way joins outperformed two-way joins for the test queries shown in Figure 6(a). The cascade of two-way joins processes the join queries in two MapReduce jobs, which means that they must write the intermediate results of the first join to HDFS, before reading them from HDFS. Furthermore, there are additional costs of initializing and cleaning up a job. In two-way and multi-way joins, our MFR-Join methods with filters delivered better performance than the basic join methods without filters. This was because large numbers of redundant intermediate results from the `partsupp` dataset, which had no selection predicate, were filtered out by the MFR-Join, as shown in Figure 6(b).

Figure 7 shows the experimental results obtained with our three-way MFR-Join using the filtering pattern. In Figure 7(a), the star-dim pattern delivered the best performance compared with the other patterns using the test queries. This was because the fact table, `partsupp`, was much larger than the dimension tables, `supplier` and `part`, in the test datasets. As shown in Figure 7(b), the number of intermediate results decreased most with the star-dim pattern. In particular, it should be noted that the star-fact pattern did not decrease the number of intermediate results at all. The join attributes in the queries were the foreign keys in the databases. Furthermore, no selection predicate was specified for the `partsupp` dataset in our test queries, so the records in `partsupp` did not play a role in filtering. The increase in the amount with a c_p value of 15 was caused by the difference in the replication factors with the best execution time. Thus, the execution times were increased slightly by creating, merging, and probing the filters needlessly. We consider that each filtering pattern may be effective in different cases, depending on the sizes of the input datasets and the ratios of joined records. Therefore, it is important to apply the filters using an advantageous pattern. If our methods are combined with upper-layer data warehouse systems, such as Hive [20], this

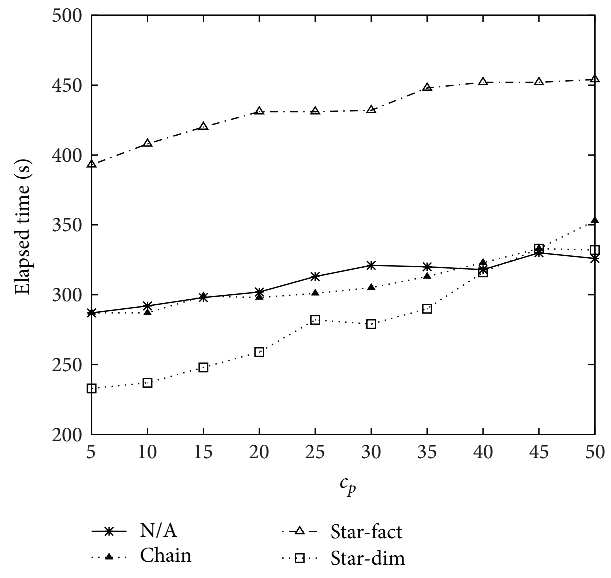


(a) Execution time

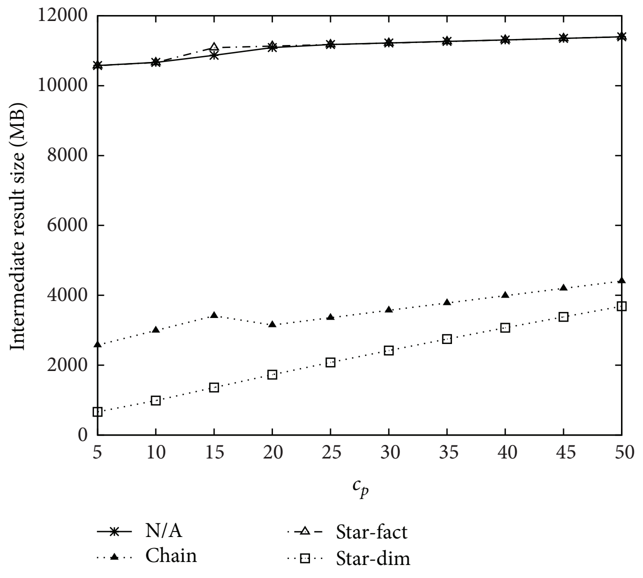


(b) Intermediate result sizes

FIGURE 6: Performance of distinct attribute joins.



(a) Execution time



(b) Intermediate result sizes

FIGURE 7: Performance of distinct attribute joins with the filtering pattern.

could be determined using its optimizer module based on statistical information related to the stored tables. This task will be addressed in our future work.

6. Conclusions

In this study, we developed methods to improve the performance of multi-way joins by applying filters. A set of filters is created and applied in turn to achieve common attribute joins and multiple sets of filters are used in various patterns, which depend on the processing order of input datasets, thereby producing distinct attribute joins. We also provide specific details for assigning reducers and writing map/reduce functions using our basic framework. We compared our proposed approach with basic multiway joins and the cascade of two-way joins. The experimental results showed that our approach improves the execution time significantly by reducing the amount of intermediate results when small portions of input datasets are joined. In future work, we plan to integrate our framework with data warehouse systems that provide query languages, such as Hive, and we aim to modify the optimizer module that supports multi-way joins to exploit filters using an appropriate pattern.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) Grant funded by the Korea Government (MSIP) (no. 20120005695), by the National Research Foundation of Korea (NRF) Grant funded by the Korea Government (MEST) (No. NRF-2009-0078828), and by Samsung Electronics Co. Ltd.

References

[1] 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 '04)*, pp. 137–150, 2004.

[2] K. Palla, *A comparative analysis of join algorithms using the hadoop map/reduce framework [M.S. thesis]*, University of Edinburgh, 2009.

[3] P. Koutris, *Bloom Filters in Distributed Query Execution*, University of Washington, 2011, http://www.cs.washington.edu/.

[4] C. Zhang, L. Wu, and J. Li, "Efficient processing distributed joins with bloomfilter using mapreduce," *International Journal of Grid and Distributed Computing*, vol. 6, no. 3, pp. 43–58, 2013.

[5] W. Li, K. Huang, D. Zhang, and Z. Qin, "Accurate counting bloom filters for large-scale data processing," *Mathematical Problems in Engineering*, vol. 2013, Article ID 516298, 11 pages, 2013.

[6] T. Lee, K. Kim, and H. J. Kim, "Join processing using bloom filter in mapreduce," in *Proceedings of the ACM Research in Applied Computation Symposium (RACS '12)*, pp. 100–105, 2012.

[7] T. Lee, K. Kim, and H. J. Kim, "Exploiting bloom filters for efficient joins in mapreduce," *Information an International Interdisciplinary Journal*, vol. 16, no. 8, pp. 5869–5885, 2013.

[8] R. Lawrence, "Using slice join for efficient evaluation of multiway joins," *Data and Knowledge Engineering*, vol. 67, no. 1, pp. 118–139, 2008.

[9] Hadoop, http://hadoop.apache.org/.

[10] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, "Parallel data processing with MapReduce: a survey," *SIGMOD Record*, vol. 40, no. 4, pp. 11–20, 2011.

[11] T. White, *Hadoop: The Definitive Guide*, O'Reilly Media, 2nd edn edition, 2011.

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

[13] L. F. Mackert and G. M. Lohman, "R^* Optimizer validation and performance evaluation for distributed queries," in *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB '86)*, pp. 149–159, August 1986.

[14] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[15] 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: Advances in Database Technology (EDBT '10)*, pp. 99–110, March 2010.

[16] 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*, vol. 23, no. 9, pp. 1299–1311, 2011.

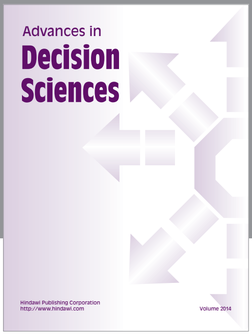
[17] A. Kemper, D. Kossmann, and C. Wiesner, "Generalized hash teams for join and group-by," in *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*, pp. 30–41, 1999.

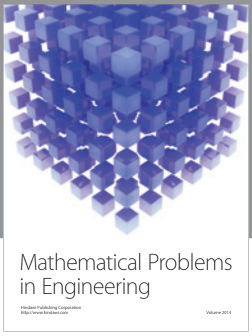
[18] C. Zhang, J. Li, L. Wu, M. Lin, and W. Liu, "Sej: an even approach to multiway theta-joins using mapreduce," in *Proceedings of the 2nd International Conference on Cloud and Green Computing (CGC '12)*, pp. 73–80, 2012.

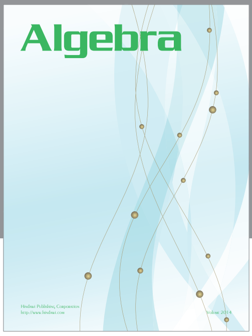
[19] TPC-H benchmark, http://www.tpc.org/tpch/.

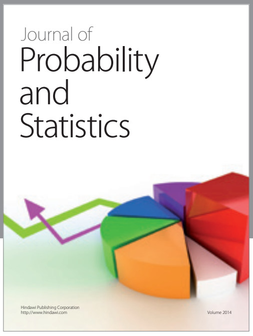
[20] A. Thusoo, J. S. Sarma, N. Jain et al., "Hive—a petabyte scale data warehouse using hadoop," in *Proceedings of the 26th IEEE International Conference on Data Engineering (ICDE '10)*, pp. 996–1005, March 2010.

Advances in Operations Research

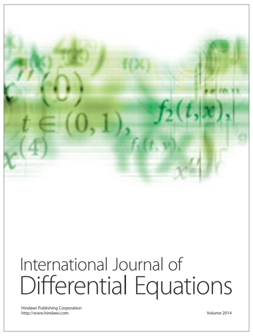

Advances in Decision Sciences


Mathematical Problems in Engineering

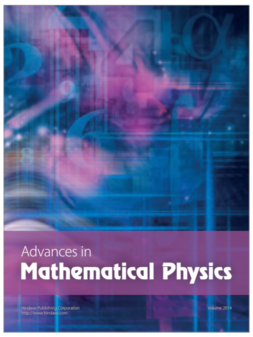

Algebra

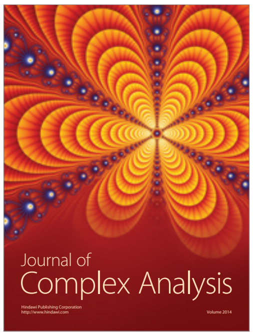

Journal of Probability and Statistics

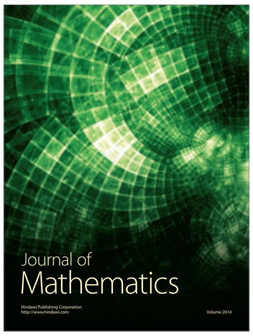
The Scientific World Journal

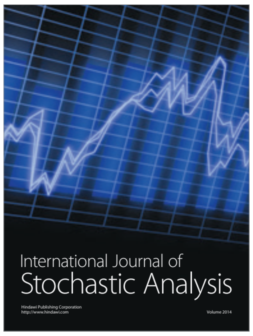

International Journal of Differential Equations

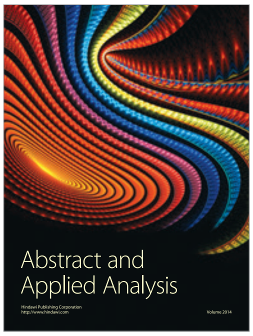
International Journal of Combinatorics

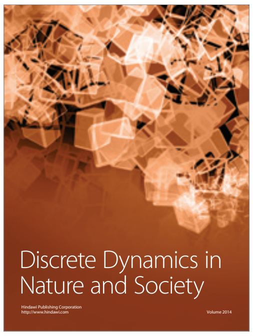

Advances in Mathematical Physics

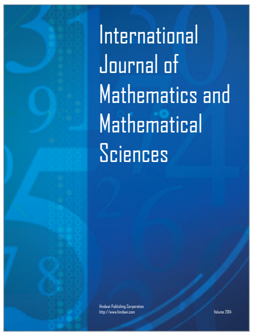

Journal of Complex Analysis


Journal of Mathematics


International Journal of Stochastic Analysis

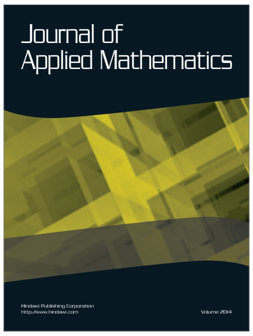

Abstract and Applied Analysis

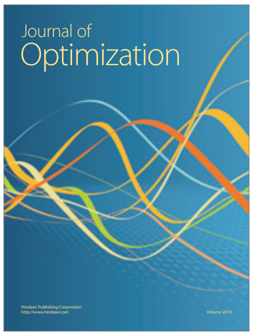

Discrete Dynamics in Nature and Society


International Journal of Mathematics and Mathematical Sciences

Journal of Discrete Mathematics

Journal of Function Spaces


Journal of Applied Mathematics


Journal of Optimization