FiDoop: Parallel Mining of Frequent Itemsets Using MapReduce

Yaling Xun, Jifu Zhang, and Xiao Qin, *Senior Member, IEEE*

***Abstract*—Existing parallel mining algorithms for frequent itemsets lack a mechanism that enables automatic parallelization, load balancing, data distribution, and fault tolerance on large clusters. As a solution to this problem, we design a parallel frequent itemsets mining algorithm called FiDoop using the MapReduce programming model. To achieve compressed storage and avoid building conditional pattern bases, FiDoop incorporates the frequent items ultrametric tree, rather than conventional FP trees. In FiDoop, three MapReduce jobs are implemented to complete the mining task. In the crucial third MapReduce job, the mappers independently decompose itemsets, the reducers perform combination operations by constructing small ultrametric trees, and the actual mining of these trees separately. We implement FiDoop on our in-house Hadoop cluster. We show that FiDoop on the cluster is sensitive to data distribution and dimensions, because itemsets with different lengths have different decomposition and construction costs. To improve FiDoop's performance, we develop a workload balance metric to measure load balance across the cluster's computing nodes. We develop FiDoop-HD, an extension of FiDoop, to speed up the mining performance for high-dimensional data analysis. Extensive experiments using real-world celestial spectral data demonstrate that our proposed solution is efficient and scalable.**

***Index Terms*—Frequent itemsets, frequent items ultrametric tree (FIU-tree), Hadoop cluster, load balance, MapReduce.**

I. Introduction

FREQUENT itemsets mining (FIM) is a core problem in association rule mining (ARM), sequence mining, and the like. Speeding up the process of FIM is critical and indispensable, because FIM consumption accounts for a significant portion of mining time due to its high computation and input/output (I/O) intensity. When datasets in modern data mining applications become excessively large, sequential FIM algorithms running on a single machine suffer from performance deterioration. To address this issue, we investigate how to perform FIM using MapReduce—a widely adopted programming model for processing big datasets by exploiting the parallelism among computing nodes of a cluster. We show how to distribute a large dataset over the cluster to balance load across all cluster nodes, thereby optimizing the performance of parallel FIM.

Frequent itemsets mining algorithms can be divided into two categories [1], [2], namely, *Apriori* and FP-growth schemes. *Apriori* is a classic algorithm using the generate-and-test process that generates a large number of candidate itemsets; *Apriori* has to repeatedly scan an entire database [3]. To reduce the time required for scanning databases, Han *et al.* [4] proposed a novel approach called FP-growth, which avoids generating candidate itemsets. Most previously developed parallel FIM algorithms were built upon the *Apriori* algorithm [5]–[8]. Unfortunately, in *Apriori*-like parallel FIM algorithms, each processor has to scan a database multiple times and to exchange an excessive number of candidate itemsets with other processors. Therefore, *Apriori*-like parallel FIM solutions suffer potential problems of high I/O and synchronization overhead, which make it strenuous to scale up these parallel algorithms. The scalability problem has been addressed by the implementation of a handful of FP-growth-like parallel FIM algorithms [9]–[12]. A major disadvantage of FP-growth-like parallel algorithms, however, lies in the infeasibility to construct in-memory FP trees to accommodate large-scale databases. This problem becomes more pronounced when it comes to massive and multidimensional databases.

Rather than considering *Apriori* and FP-growth, we incorporate the frequent items ultrametric tree (FIU-tree) in the design of our parallel FIM technique. We focus on FIU-tree because of its four salient advantages, which include reducing I/O overhead, offering a natural way of partitioning a dataset, compressed storage, and averting recursively traverse [13]. More importantly, the existing parallel algorithms lack a mechanism that enables automatic parallelization, load balancing, data distribution, and fault tolerance on large computing clusters. To solve the aforementioned open problems, we design a parallel FIM algorithm called FiDoop using the MapReduce programming model (see [14]–[17] for details on MapReduce).

Compared with the existing frequent items ultrametric tree (FIUT) algorithm [13], FiDoop has distinctive features. In FiDoop, the mappers independently and concurrently decompose itemsets; the reducers perform combination operations by constructing small ultrametric trees as well as mining

Manuscript received December 2, 2014; accepted April 7, 2015. Date of publication June 15, 2015; date of current version February 12, 2016. This work was supported by the National Natural Science Foundation of China under Grant 61272263. The work of X. Qin was supported by the U.S. National Science Foundation under Grant CCF-0845257 (career development), Grant CNS-0917137 [computer system research (CSR)], Grant CNS-0757778 (CSR), Grant CCF-0742187 (computing processes and artifacts), Grant CNS-0831502 (CyberTrust), Grant CNS-0855251 (computing research infrastructure), Grant OCI-0753305 (Cyberinfrastructure-TEAM), Grant DUE-0837341 (course, curriculum, and laboratory improvement), and Grant DUE-0830831 (scholarship for service). This paper was recommended by Associate Editor J. Lu. *(Corresponding author: Jifu Zhang.)*

Y. Xun and J. Zhang are with the Taiyuan University of Science and Technology, Taiyuan 030024, China (e-mail: zjf@tyust.edu.cn).

X. Qin is with the Department of Computer Science and Software Engineering, Samuel Ginn College of Engineering, Auburn University, Auburn, AL 36849-5347 USA (e-mail: xqin@auburn.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TSMC.2015.2437327

2168-2216 © 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:32:48 UTC from IEEE Xplore. Restrictions apply.

these trees in parallel. We implement FiDoop on our in-house Hadoop cluster. We observe that data partitioning and distribution are critical issues in FiDoop, because itemsets with different lengths have various decomposition and construction costs. To optimize the performance of FiDoop, we propose a new data partitioning method to well balance computing load among the cluster nodes; we develop FiDoop-HD, an extension of FiDoop, to meet the needs of high-dimensional data processing.

The main contributions of this paper are summarized as follows.

1) We made a complete overhaul to FIUT (i.e., the frequent items ultrametric trees method), and addressed the performance issues of parallelizing FIUT.
2) We developed the parallel frequent itemsets mining method (i.e., FiDoop) using the MapReduce programming model.
3) We proposed a data distribution scheme to balance load among computing nodes in a cluster.
4) We further optimized the performance of FiDoop and reduced running time of processing high-dimensional datasets.
5) We conducted extensive experiments using a wide range of synthetic and real-world datasets, and we show that FiDoop is efficient and scalable on Hadoop clusters.

The remainder of this paper is organized as follows. Section II describes the background knowledge. Section III gives an overview of the process of FiDoop on MapReduce. Section IV presents the design issues of FiDoop built on the MapReduce framework, followed by the implementation details are discussed in Section V, in which we pay particular attention to the performance optimization of the last MapReduce job. Section VI implements dimension reduction techniques (FiDoop-HD) to optimize FiDoop. Section VII evaluates the performance of FiDoop on a real-world cluster. Section VIII discusses the related work. Finally, Section IX concludes this paper.

II. Preliminary

In this section, we first briefly review association rules. Then, we summarize the basic idea of the FIUT algorithm along with its core data structures. To facilitate the presentation of FiDoop, we introduce the MapReduce programming framework.

A. Association Rules

ARM provides a strategic resource for decision support by extracting the most important frequent patterns that simultaneously occur in a large transaction database. A typical ARM application is market basket analysis. An association rule, for example, can be "if a customer buys A and B, then 90% of them also buy C." In this example, 90% is the confidence of the rule.

Apart from confidence, support is another measure of association rules, each of which is an implication in the form of $X \Rightarrow Y$. Here, X and Y are two itemsets, and $X \cap Y = \emptyset$. The confidence of a rule $X \Rightarrow Y$ is defined as a ratio between support$(X \cup Y)$ and support(X). Note that, an itemset X has support s if s% of transactions contain the itemset. We denote $s = \text{support}(X)$; the support of the rule $X \Rightarrow Y$ is support$(X \cup Y)$.

The ultimate objective of ARM is to discover all rules that satisfy a user-specified minimum support and minimum confidence. The ARM process can be decomposed into two phases: 1) identifying all frequent itemsets whose support is greater than the minimum support and 2) forming conditional implication rules among the frequent itemsets. The first phase is more challenging and complicated than the second one. As such, most prior studies are primarily focused on the issue of discovering frequent itemsets.

B. FIUT

The FIUT approach adopts the FIU-tree to enhance the efficiency of mining frequent itemsets. FIU-tree is a tree structure constructed as follows.

1) After the root is labeled as null, an itemset $p_1, p_2, \ldots, p_m$ of frequent items is inserted as a path connected by edges $(p_1, p_2), (p_2, p_3), \ldots, (p_{m-1}, p_m)$ without repeating nodes, beginning with child p_1 of the root and ending with leaf p_m in the tree.
2) An FIU-tree is constructed by inserting all itemsets as its paths, each itemset contains the same number of frequent items. Thus, all of the FIU-tree leaves are identical height.
3) Each leaf in the FIU-tree is composed of two fields: named item-name and count. The count of an item-name is the number of transactions containing the itemset that is the sequence in a path ending with the item-name. Nonleaf nodes in the FIU-tree contains two fields: named item-name and node-link. A node-link is a pointer linking to child nodes in the FIU-tree.

The FIUT algorithm consists of two key phases. The first phase involves two rounds of scanning a database. The first scan generates frequent one-itemsets by computing the support of all items, whereas the second scan results in k-itemsets by pruning all infrequent items in each transaction record. Note that, k denotes the number of frequent items in a transaction. In phase two, a k-FIU-tree is repeatedly constructed by decomposing each h-itemset into k-itemsets, where $k + 1 \leq h \leq M$ (M is the maximal value of k), and unioning original k-itemsets. Then, phase two starts mining all frequent k-itemsets based on the leaves of k-FIU-tree without recursively traversing the tree. Compared with the FP-growth method, FIUT significantly reduces the computing time and storage space by averting overhead of recursively searching and traversing conditional FP trees.

C. MapReduce Framework

MapReduce is a promising parallel and scalable programming model for data-intensive applications and scientific analysis. A MapReduce program expresses a large distributed computation as a sequence of parallel operations on datasets of key/value pairs. A MapReduce computation has two phases, namely, the Map and Reduce phases. The Map phase splits

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:32:48 UTC from IEEE Xplore. Restrictions apply.

TABLE I
SYMBOL AND ANNOTATION

Symbol	Annotation
minsupport	User-specified Minimum Support threshold
k − itemsets	Itemsets containing k items
k − FIU − tree	FIU-tree constructed by all k-itemsets
M	The maximum value of k
IS_m	Itemsets in which the length of each itemset is m

the input data into a large number of fragments, which are evenly distributed to Map tasks across the nodes of a cluster to process. Each Map task takes in a key-value pair and then generates a set of intermediate key-value pairs. After the MapReduce runtime system groups and sorts all the intermediate values associated with the same intermediate key, the runtime system delivers the intermediate values to Reduce tasks. Each Reduce task takes in all intermediate pairs associated with a particular key and emits a final set of key-value pairs. Both input pairs of Map and the output pairs of Reduce are managed by an underlying distributed file system. MapReduce greatly improves programmability by offering automatic data management, highly scalable, and transparent fault-tolerant processing. Also, MapReduce is running on clusters of cheap commodity servers—an increasingly attractive alternative to expensive computing platforms. Thanks to the aforementioned advantages, MapReduce has been widely adopted by companies like Google, Yahoo, Microsoft, and Facebook.

Hadoop—one of the most popular MapReduce implementations—is running on clusters where Hadoop distributed file system (HDFS) stores data to provide high aggregate I/O bandwidth. At the heart of HDFS is a single NameNode—a master server that manages the file system namespace and regulates access to files. The Hadoop runtime system establishes two processes called JobTracker and TaskTracker. JobTracker is responsible for assigning and scheduling tasks; each TaskTracker handles Map or Reduce tasks assigned by JobTracker.

III. OVERVIEW OF FIDOOP

In light of the MapReduce programming model, we design a parallel frequent itemsets mining algorithm called FiDoop. The design goal of FiDoop is to build a mechanism that enables automatic parallelization, load balancing, and data distribution for parallel mining of frequent itemsets on large clusters. To facilitate the presentation of FiDoop, we summarize the notation used throughout this paper in Table I.

Aiming to improve data storage efficiency and to avert building conditional pattern bases, FiDoop incorporates the concept of FIU-tree rather than traditional FP trees. We observe that parallelizing the serial algorithm FIUT is a challenge [see also the main program of the FIUT algorithm in Algorithm 1(A)]. After *h*-itemsets are generated in phase 1, an iterative process is repeatedly running to construct *k*-FIU trees and to discover frequent *k*-itemsets until the *k* value is from *M* to 2. In other words, the construction of *k*-FIU trees and the discovery of frequent *k*-itemsets are executed in a sequential way. Even worse, it is nontrivial to construct *k*-FIU-tree, which is the most important and time consuming phase. The *k*-FIU-tree-generation algorithm (*h*-itemsets) [see Algorithm 1(B)] constructs a *k*-FIU tree by decomposing each *h*-itemset into *k*-itemsets, where $k+1 \le h \le M$. Then, the union of original *k*-itemsets is calculated to construct the *k*-FIU tree. The generation of the *k*-itemsets requires decomposing all possible *h*-itemsets ($h > k$); thus, the decomposition process is sequentially performed starting from long to short itemsets. As such, we improve the serial FIUT algorithm as follows.

1) The first phase of FIUT involving two rounds of scanning a database is implemented in the form of two MapReduce jobs. The first MapReduce job is responsible for the first round of scanning to create frequent one-itemsets. The second MapReduce job scans the database again to generate *k*-itemsets by removing infrequent items in each transaction.
2) The second phase of FIUT involving the construction of a *k*-FIU tree and the discovery of frequent *k*-itemsets is handled by a third MapReduce job, in which *h*-itemsets ($2 \le h \le M$) are directly decomposed into a list of $(h-1)$-itemsets, $(h-2)$-itemsets, ..., and two-itemsets. In the third MapReduce job, the generation of short itemsets is independent to that of long itemsets. In other words, long and short itemsets are created in parallel by our parallel algorithm. Such an optimization approach solves the parallelization problem of Algorithms 1(A) and (B).

The three MapReduce jobs of our proposed FiDoop are described in detail.

The first MapReduce job discovers all frequent items or frequent one-itemsets (see Algorithm 2). In this phase, the input of Map tasks is a database, and the output of Reduce tasks is all frequent one-itemsets. The second MapReduce job scans the database to generate *k*-itemsets by removing infrequent items in each transaction (see Algorithm 3). The last MapReduce job—the most complicated one of the three—constructs *k*-FIU-tree and mines all frequent *k*-itemsets. In this paper,

Algorithm 1 FIUT

```
1: function ALGORITHM 1(A): FIUT( D, n)
2:     h-itemsets = k-itemsets generation(D, MinSup);
3:     for k = M down to 2 do
4:         k-FIU-tree = k-FIU-tree generation (h-itemsets);
5:         frequent k-itemsets Lk = frequent k-itemsets generation (k-FIU-tree);
6:     end for
7: end function
8: function ALGORITHM 1(B): K-FIU-TREE GENERATION((h-itemsets))
9:     Create the root of a k-FIU-tree, and label it as null (temporary 0th root)
10:    for all (k + 1 ≤ h ≤ M) do
11:        decompose each h-itemset into all possible k-itemsets, and union original k-itemsets;
12:        for all (k-itemset) do
13:            ··· build k-FIU-tree( ); here, pseudo code is omitted;
14:        end for
15:    end for
16: end function
```

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:32:48 UTC from IEEE Xplore. Restrictions apply.

Algorithm 2 ParallelCounting: To Generate All Frequent One-Itemsets

```
Input: minsupport, DBi;
Output: 1-itemsets;
 1: function MAP(key offset, values DBi)
 2:     //T is the transaction in DBi
 3:     for all T do
 4:         items ← split each T;
 5:         for all item in items do
 6:             output( item, 1);
 7:         end for
 8:     end for
 9: end function
10: reduce input: (item,1 )
11: function REDUCE(key item, values 1)
12:     sum=0;
13:     for all item do
14:         sum += 1;
15:     end for
16:     output(1-itemset, sum); //item is stored as 1-itemset
17:     if sum ≥ minsupport then
18:         F − list ← the (1-itemset, sum) //F-list is a CacheFile storing
    frequent 1-itemsets and their count.
19:     end if
20: end function
```

Algorithm 3 Generatekitemsets: To Generate All k-Itemsets by Pruning the Original Database

```
Input: minsupport, DBi;
Output: k-itemsets;
 1: function MAP(key offset, values DBi)
 2:     //T is the transaction in DBi
 3:     for all (T) do
 4:         items ← split each T;
 5:         for all (item in items) do
 6:             if (item is not frequent) then
 7:                 prune the item in the T;
 8:             end if
 9:             k-itemset ←(k, itemset) /*itemset is the set of frequent items
    after pruning, whose length is k */
10:             output(k-itemset,1);
11:         end for
12:     end for
13: end function
14: function REDUCE(key k-itemset, values 1)
15:     sum=0;
16:     for all (k-itemset) do
17:         sum += 1;
18:     end for
19:     output(k, k-itemset+sum);//sum is support of this itemset
20: end function
```

we pay particular attention to the third MapReduce job, which is a performance bottleneck of the FiDoop algorithm.

At the core of FiDoop is the third MapReduce job, which is in charge of decomposing, building an FIUT tree, and mining the FIUT tree. Let us outline the process of the third MapReduce job by considering a case where each itemset has k items. Note that, the initial value of k is set to M. First, FiDoop handles a set of mappers, each of which repeatedly decomposes h-itemsets ($2 < h \leq M$) into a list of $(h-1)$-itemsets, $(h-2)$-itemsets, ..., and two-itemsets. Applying multiple mappers to decompose h-itemsets in parallel improves data storage efficiency and I/O performance. Then, FiDoop leverages multiple reducers to merge itemsets with the same item number including original itemsets and to construct a k-FIU-tree from the itemsets decomposed and

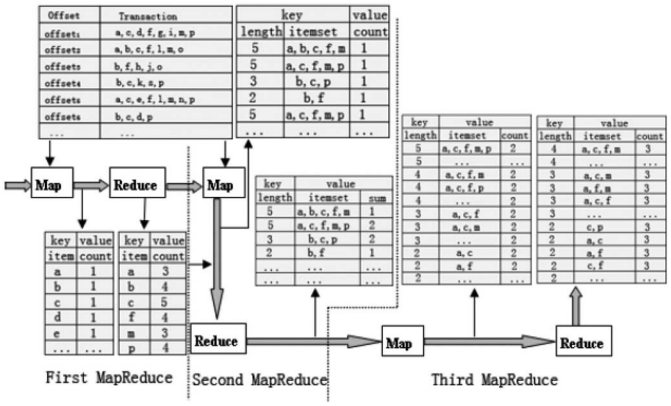


Fig. 1. Overview of MapReduced-based FiDoop.

emitted by the mappers. FiDoop's tree-construction procedure takes full advantage of the Hadoop runtime system, in which the shuffling phase copies result pairs with a same key generated by mappers to a reducer. During the shuffling phase of FiDoop's third MapReduce job, we choose item numbers as output keys of key-value pairs emitted by the mappers. Finally, the k-FIU-tree is mined without necessity to mine recursively; the k-FIU-tree is immediately discarded after frequent k-itemsets are mined.

From the aforementioned description of the processes, we show that frequent one-itemsets are directly generated by scanning a database. FiDoop carries out a two-stage process to construct a k-FIU-tree ($2 \leq k \leq M$) from k-itemsets. In the first stage, k-itemsets are obtained by pruning infrequent items of each transaction in the second scan for database. The second stage is the combination of k-itemsets generated by decomposing all h-itemsets ($k < h$). Please note that the two-stage process is similar to that of the FIUT algorithm, which ensures the correctness of our algorithm.

The following preliminary findings motivate us to address a pressing issue pertinent to balancing load in FiDoop: 1) large itemsets give rise to high-decomposition overhead and 2) and small decomposed itemsets lead to a large number of itemsets. To achieve good load balancing performance, we incorporate constraints in the shuffling phase of the MapReduce jobs in FiDoop, thereby balancing the number of itemsets across reducers (see Section V-A for details on load balancing).

IV. MapReduce-Based FiDoop

In this section, we present the design issues of FiDoop built on the MapReduce framework. Fig. 1 depicts the working flow of FiDoop consisting of three MapReduce jobs. Recall that the intermediate results provided by the mappers in the third MapReduce job are used to construct FIU trees (see Algorithm 4). We intentionally keep such intermediate key-value pairs output from Fig. 1 to make the process flow concise.

A. First MapReduce Job

The first MapReduce job is responsible for creating all frequent one-itemsets. A transaction database is partitioned into

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:32:48 UTC from IEEE Xplore. Restrictions apply.

Algorithm 4 Miningkitemsets: To Mine All Frequent Itemsets

Input: Pair(k, k-itemset+support);//This is the output of the second MapReduce.
Output: frequent k-itemsets;
1: **function** MAP(key k, values k-itemset+support)
2: De-itemset ← values.k-itemset;
3: *decompose*(De-itemset,2,mapresult); /* To decompose each De-itemset into t-itemsets (t is from 2 to De-itemset.length), and store the results to mapresult. */
4: **for all** (mapresult with different item length) **do**
5: //t-itemset is the results decomposed by k-itemset(i.e. t ≤ k);
6: **for all** (t-itemset) **do**
7: $t - FIU - tree$ ← t-FIU-tree generation(local-FIU-tree, t-itemset);
8: output(t, t-FIU-tree);
9: **end for**
10: **end for**
11: **end function**
12: **function** REDUCE(key t, values t-FIU-tree)
13: **for all** (t-FIU-tree) **do**
14: $t - FIU - tree$ ← combining all t-FIU-tree from each mapper;
15: **for all** (each leaf with item name v in t-FIU-tree) **do**
16: **if** (count(v)/| DB |≥ minsupport) **then**
17: frequent $h - itemset$ ← $pathitem(v)$;
18: **end if**
19: **end for**
20: **end for**
21: **output(h, frequent h-itemset);**
22: **end function**

multiple input files stored by the HDFS across data nodes of a Hadoop cluster. Each mapper sequentially reads each transaction from its local input split, where each transaction is stored in the format of pair<LongWritable offset, Text record>. Then, mappers compute the frequencies of items and generate local one-itemsets. Next, these one-itemsets with the same key emitted by different mappers are sorted and merged in a specific reducer, which further produces global one-itemsets. Finally, infrequent items are pruned by applying the minsupport; and consequently, global frequent one-itemsets are generated and written in the form of pair<Text item, LongWritable count> as the output from the first MapReduce job. Importantly, frequent one-itemsets along with their counts are stored in a local file named *F*-list, which becomes the input of the second MapReduce job in FiDoop. The pseudocode of the first MapReduce job is detailed in Algorithm 2.

B. Second MapReduce Job

Given frequent one-itemsets generated by the first MapReduce job, the second MapReduce job applies a second-round of scanning on the database to prune infrequent items from each transaction record. The second job marks an itemset as a k-itemset if it contains k frequent items ($2 \leq k \leq M$, where M is the maximal value of k in the pruned transactions).

Each mapper of the second job takes transactions as input. Then, the mapper emits a set of pair <ArrayWritable itemsets, Longwritable ONE>, in which itemsets is composed of the number of the items produced by pruning and the set of items. These pairs obtained by the second MapReduce job's mappers are combined and shuffled for the second job's reducers.

After performing the combination operation, each reducer emits key/value pairs, where the key is the number of each itemset and the value is each itemset and its count. More formally, the output of the second MapReduce job is pair<IntWritable itemnumber, MapWritable<ArrayWritable k-item, LongWritable SUM>>. Algorithm 3 outlines the pseudocode of the second job's Map and Reduce functions. It is important to ensure that frequent items in each transaction should retain their lexicographical order in order to facilitate the next phase.

C. Third MapReduce Job

The third MapReduce job—a computationally expensive phase—is dedicated to: 1) decomposing itemsets; 2) constructing k-FIU trees; and 3) mining frequent itemsets.

The main goal of each mapper is twofold: 1) to decompose each k-itemset obtained by the second MapReduce job into a list of small-sized sets, where the number of each set is anywhere between 2 to $k-1$ and 2) to construct an FIU-tree by merging local decomposition results with the same length.

The third MapReduce job is highly scalable, because the decomposition procedure of each mapper is independent of the other mappers. In other words, the multiple mappers can perform the decomposition process in parallel. The local procedure of constructing an FIU-tree is described in Section II-B. Such an FIU-tree construction improves data storage efficiency and I/O performance; the improvement is made possible thanks to merging the same itemsets in advance using small FIU trees.

The Map function of the third job generates a set of key/value pairs, in which the key is the number of items in an itemset and the value is an FIU-tree that is comprised of nonleaf and leaf nodes. Nonleaf nodes include item-name and node-link; leaf nodes include item-name and its support. In doing so, itemsets with the same number of items are delivered to a single reducer. By parsing the key-value pair ($k2$, $v2$), the reducer is responsible for constructing $k2$-FIU-tree and mining all frequent itemsets only by checking the count value of each leaf in the $k2$-FIU-tree without repeatedly traversing the tree.

Algorithm 4 illustrates the Map and Reduce functions. Here, the details on the function of t-FIU-tree generation (t-itemset) can be found in [13]. In Algorithm 4, the *decompose()* function is a recursive one, decomposing an h-itemset into a list of k-itemsets, where k is an integer between 2 and h. Our implementation of the *decompose()* function is shown in Algorithm 5.

V. Implementation Details

Now, we discuss the implementation details of FiDoop. We pay particular attention to the last MapReduce job in FiDoop, because the last job is computationally expensive. We show how to optimize the performance of the third MapReduce job in two approaches.

A. Load Balance

The *decompose*() function of the third MapReduce job accomplishes the decomposition process. If the length of an itemset is m, the time complexity of decomposing the itemset is $O(2^m)$. Thus, the decomposition cost is exponentially proportional to the itemset's length. In other words, when the itemset length is going up, the decomposition overhead will

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:32:48 UTC from IEEE Xplore. Restrictions apply.

Algorithm 5 Decompose(): How to Decompose a k-Itemset

```
Input: to be decomposed string s;
Output: the decomposed results l;
 1: function DECOMPOSE(s, l, de-result)
 2:     /* s is the string to be decomposed, l is the length of the itemset
    required to be decomposed, de-result stores the results.
 3:     for all ( i is from l to s.length) do
 4:         decompose(s, i, result, resultend);
 5:         de-result ← i+resultend;
 6:     end for
 7: end function
 8: function DECOMPOSE(s, m, result, resultend)
 9:     if (m == 0) then
10:         resultend.addAll(result);//resultend is a list storing all the i-item
    set
11:         return;
12:     end if
13:     if (s is not null) then
14:         result.add(s[0]+null);
15:         for all (j is from the second value to the last value of s) do
16:             s1[j − 1] ← s[j];
17:         end for
18:         decompose(s1, m - 1, result, resultend); //when selecting the first
    item
19:         result.remove(result.size() - 1);
20:         desompose(s1, m, result, resultend); //when the first item is not
    selected
21:     end if
22: end function
```

dramatically enlarged. The data skewness problem is mainly induced by the decomposition operation, which in turn has a significant performance impact on FiDoop.

The first step toward balancing load among data nodes of a Hadoop cluster is to quantitatively measure the total computing load of processing local itemsets. We achieve this first step by developing a workload-balance metric to quantify load balance among the data nodes.

We consider a database partitioned across p data nodes. Let $p_i(\mathrm{IS}_m) = (C_i(\mathrm{IS}_m)/\sum_{j=1}^{p} C_j(\mathrm{IS}_m))$ be the probability that node i contains itemsets whose length is m. Here, IS_m denotes a set of itemsets where the length of each itemset is m; $C_i(\mathrm{IS}_m)$ represents the count of IS_m in node i. In what follows, we define a weight to quantify the computing load of IS_m.

Given an itemset IS_m, the computing-load weight of IS_m over all the itemsets is defined as

$$\omega(\mathrm{IS}_m) = \frac{C(\mathrm{IS}_m) \times 2^m}{\sum_{j=2}^{M} C\big(\mathrm{IS}_j\big) \times 2^j} \tag{1}$$

where 2^m is the time complexity of decomposing m-itemsets (see Algorithm 5); $C(\mathrm{IS}_m)$ is the count of IS_m over all available data nodes.

Based on the definition of $\omega(\mathrm{IS}_m)$ in (1). We define the computing load of node i.

Given a transaction database $\mathbf{D}$ partitioned over p nodes and a random itemset $\mathbf{X}$, the computing load of node i is expressed as

$$W_i = \sum_{X \subset \mathrm{IS}} \omega(X) \times p_i(X). \tag{2}$$

The summation of all the computing-load over all nodes is one; thus, we have $\sum_{j=1}^{p} W_j = 1$. A data distribution leads to high load-balancing performance if the weights W_i ($i \in [1, p]$) are identical. On the other hand, a large discrepancy of the W_i gives rise to poor load-balancing performance. We introduce the entropy measure as a load-balancing metric. Given a database $\mathbf{D}$ partitioned over p nodes, the load-balancing metric is expressed as

$$\mathrm{WB}(D) = \frac{\sum_{j=1}^{p} W_j \log\big(W_j\big)}{\log(p)}. \tag{3}$$

The $\mathrm{WB}(D)$ metric defined in the form of entropy has the following properties.

1) If $\mathrm{WB}(D)$ equals to 1 (i.e., $\mathrm{WB}(D) = 1$), decomposition load is perfectly balanced across all the nodes.
2) If $\mathrm{WB}(D)$ equals to 0 (i.e., $\mathrm{WB}(D) = 0$), decomposition load is concentrated on one node.
3) All the other cases are represented by $0 < \mathrm{WB}(D) < 1$.

We experimentally evaluate the load-balancing performance of FiDoop in Section VII-B.

B. High-Dimensional Optimization

The aforementioned analysis confirms that if the length of itemsets to be decomposed is large, the decomposition cost will exponentially increase. In this section, we conduct experiments to investigate the impact of dimensionality on FiDoop. We also compare FiDoop with a popular solution parallelization of FP-growth (Pfp) [18]. Section VI presents an optimization algorithm called FiDoop-HD for high-dimensional data processing.

When it comes to mining frequent itemsets, varying dimensionality leads to a wide range of itemset lengths. Our algorithm needs to decompose each itemset generated by pruning infrequent items for each transaction. Fig. 2 shows the impact of dimensionality on the processing time of the tested algorithms. We made use of the series of D1000W, which are described in detail in Section VII (see Synthetic Dataset). In the group of experiments, the number of transactions is 10 000 000 and the average transaction size is anywhere between 10 and 50.

Fig. 2(a) demonstrates that the running times of FiDoop and Pfp sharply go up when the number of dimensions increases. In other words, both approaches are heavily sensitive to the number of dimensions. When the number of dimensions is small, FiDoop is faster than Pfp thanks to the fact that FiDoop can avert building conditional pattern bases and conditional sub-FP trees for short patterns. Pfp has poor performance, because it has to recursively traverse conditional FP trees. Furthermore, in order to facilitate parallelism, Pfp groups frequent one-itemsets and distributes the data corresponding to these items to each computing node; such a grouping strategy is both time and space consuming. Nevertheless, when the dimensionality approximately reaches 30, FiDoop's performance starts degrading. This is because the cost of decomposing a k-itemset is very expensive (i.e., 2^m, m is determined by the dimensionality of the dataset). The increasing value of the m exponentially enlarges the running time of FiDoop. To address this performance issue, we propose in Section VI an optimization approach to boost the speed of processing high-dimensional data.

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:32:48 UTC from IEEE Xplore. Restrictions apply.

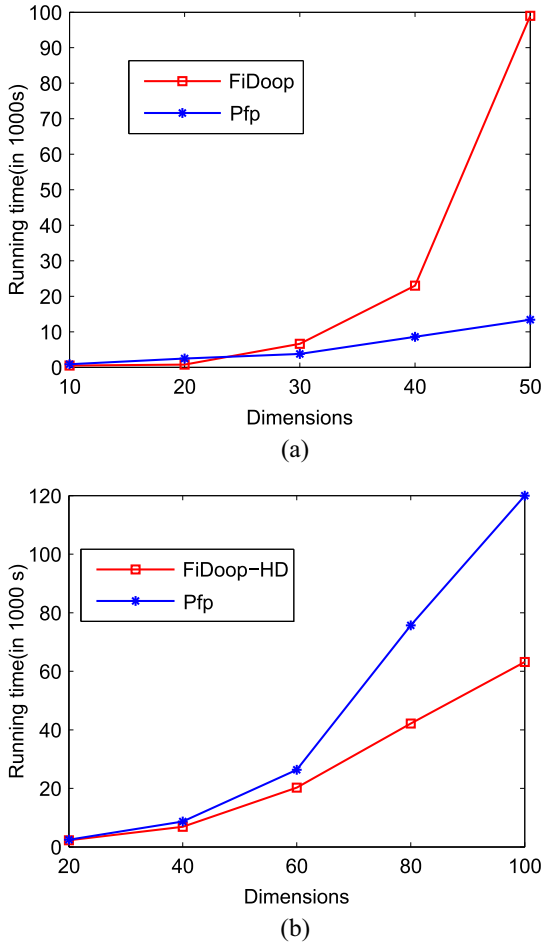


Fig. 2. Effect of dimensionality on different algorithms (on four nodes). Comparison of (a) FiDoop and Pfp and (b) FiDoop-HD and Pfp.

VI. FiDoop-HD

Recognizing that FiDoop exhibits high performance for low-dimensional databases, we design and implement a dimensionality reduction scheme to efficiently handle high-dimensional data processing. This approach called FiDoop-HD is an extension of the FiDoop algorithm. FiDoop-HD carries out the following steps to judiciously process high-dimensional data.

1) The output data of the second MapReduce job in FiDoop are, respectively, stored in multiple cache files according to itemset lengths. Thus, all k-itemsets are recorded in a file named k-file, whereas all $(k-1)$-itemsets are written to another file named $(k-1)$-file.
2) FiDoop-HD decomposes the list of itemsets in a decreasing order of itemset length. After reading M-itemsets from a cache file, FiDoop-HD decomposes the M-itemsets into a list of $(M-1)$-itemsets. Note that, M is the maximal length of itemsets. Then, these itemsets combine original $(M-1)$-itemsets to be stored. Next, FiDoop-HD loads a cache file to store $(M-1)$-itemsets, which are decomposed into $(M-2)$-itemsets to be unioned into the original $(M-2)$-itemsets. This procedure is repeatedly carried out until the entire decomposition process is accomplished. The mappers of the third MapReduce job only decompose k-itemset into $(k-1)$-itemsets rather than into two-itemsets (see Algorithm 6). In case of multiple files stored on a data node, the node sequentially loads and processes the files.

Algorithm 6 FiDoop-HD-MiningMap: High-Dimensional Optimization for Map() Function

Input: k-file /*k-file($2 \le k \le M$) is used to store the frequent k-itemsets generated in the second MapReduce.*/
Output: (k-1)-FIU-tree

```
1: function MAP(key k, values k-file)
2:     for all (k is from M to 2) do
3:         for all (k-itemset in k-file) do
4:             decompose(k-itemset, k-1, (k-1)-itemsets); /*Each k-itemset is only decomposed into (k-1)-itemsets */
5:             (k-1)-file ← the decomposed (k-1)-itemsets union the original (k-1)-itemsets in (k-1)-file;
6:             for all (t-itemset in (k-1)-file) do
7:                 t − FIU − tree ← t-FIU-tree generation(local-FIU-tree, t-itemset);
8:                 output(t, t-FIU-tree);
9:             end for
10:        end for
11:    end for
12: end function
```

In Algorithm 6, the cost of decomposing an m-itemset into $(m-1)$-itemsets can be modeled as c_m^{m-1}. Given a file storing all itemsets whose length is m, the decomposition cost of the file is $C(\mathrm{IS}_m) \times c_m^{m-1}$, where $C(\mathrm{IS}_m)$ is the count of IS_m in the file. Hence, the time complexity of the entire process can be approximated as $\max(C(\mathrm{IS}_i)) \times (c_M^{M-1} + c_{M-1}^{M-2} + \cdots + c_3^2)$, which can be further written as $\max(C(\mathrm{IS}_i)) \times (M \times (M+1)/2)$, $2 < i \le M$.

The time complexity of FiDoop-HD is much lower than that of FiDoop. Such a performance improvement is evident from the experimental results plotted in Fig. 2(b). We conduct the scalability study using datasets that containing 10 000 000 transactions, in which the average transaction size varies anywhere from 20 to 100 in step 20. For example, Fig. 2(b) reveals that our solution substantially improves the performance of FiDoop. We implement FiDoop on a Hadoop cluster, where massive amounts of data are managed by HDFS. It is essential and critical to address the I/O performance issues in FiDoop-HD due to the following reasons. First, itemsets decomposed in the previous stages have to be saved in new files for subsequent phases. Second, FiDoop-HD does inherently incorporate a load-balancing policy, because each node processes the files storing itemsets with an identical length.

VII. Experimental Evaluation

We evaluate the performance of FiDoop on our in-house Hadoop cluster equipped with 16 data nodes. Each node has an Intel Pentium-4 3.0 GHz processor, 512 MB main memory, and runs on the Ubuntu 10.10 operating system, on which Java JDK 1.6.0_37 and Hadoop 1.1.1 are installed. All the data nodes in the cluster have Gigabit Ethernet network interface cards connected to Gigabit ports on the switch; the nodes can communicate with one another using the secure shell protocol. We use the default Hadoop parameter configurations to set the replication factor (i.e., 3) and the number of Map and Reduce tasks.

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:32:48 UTC from IEEE Xplore. Restrictions apply.

To evaluate the performance of the proposed FiDoop, we use both synthetic and real-world datasets in our experiments.

1) *Synthetic Dataset:* We generated a series of synthetic datasets (i.e., the D1000W datasets) using the IBM quest market-basket synthetic data generator, which can be configured to create a wide range of datasets to meet the needs of various test requirements [19]. The number of items in each D1000W dataset is set to 1000 (represents the number of varieties of goods); while conducting the experiments, we vary the average transaction size and the number of transactions in the D1000W datasets. In the empirical study, we make use of the D1000W datasets to investigate the impacts of dimensions and the data size on the performance of the tested algorithms.
2) *Celestial Spectral Dataset:* We apply FiDoop to implement a parallel data mining application for celestial spectral data. We use the real-world celestial spectral dataset to evaluate speedup, load-balancing performance, as well as the impact of minimum support. The celestial spectral dataset used in our experiments has 5 000 000 transactions and 44 dimensions.

Note that, if not specified the minsupport is set as 0.0001 and the number of nodes is on four nodes, 10- and 40-dimension in the D1000W datasets are respective the representatives of low- and high-dimensional synthetic datasets in the following set of experiments.

A. Minimum Support

Minimum support plays an important role in mining frequent itemsets. We increase minimum support thresholds from 0.0001% to 0.0003% with an increment of 0.00005%, thereby evaluating the impact of minimum support on Pfp and our proposed algorithms containing three MapReduce jobs using both celestial spectral and synthetic datasets.

Fig. 3(a)–(c) shows the execution times of the three algorithms on 10- and 40-dimensional synthetic datasets and celestial spectral dataset, respectively. In this set of experiment, we increase the minsupport from 1×10^{-4} to 3×10^{-4} with an increment of 0.5×10^{-4}. A small minimum support slows down the performance of the evaluated algorithms. This is because an increasing number of items satisfy the small minimum support when the minsupport is decreased; it takes an increased amount of time to process the large number of items. We also observe from Fig. 3(a) that the proposed algorithms are superior to Pfp in processing the low-dimensional dataset. When it comes to the high-dimensional dataset [see Fig. 3(b) and (c)], FiDoop-HD behaves optimally and FiDoop shows its downside. These performance trends are reasonable, because the decomposition cost of FiDoop will exponentially increase, which in turn gradually offsets the gain in mining capacity with the increase of the itemset length. It is evident from these experimental results that FiDoop-HD improves the performance of FiDoop in the case of high-dimensional datasets. This observation is consistent with those drawn from Fig. 3(d)–(f), which show the running time of the three stages of FiDoop, FiDoop-HD, and Pfp on celestial spectral dataset. Although Pfp is seemingly superior to FiDoop in running time when high-dimensional datasets are processed, Pfp's space consumption and the shuffling cost in the parallel process are higher than those of our solution. As a result, FiDoop-HD's performance is better than that of Pfp.

Fig. 3(d) and (e) reveals that the running time of the first and second MapReduce jobs in FiDoop and FiDoop-HD are insensitive to minimum support. The mappers in FiDoop and FiDoop-HD have to scan the entire dataset and the reducers combine the output produced by the mappers; a similar case applies for the first MapReduce job of Pfp [see Fig. 3(f)]. Interestingly, the running times the third MapReduce job of our algorithms and the second MapReduce job of Pfp sharply increase with the decreasing value of minimum support. Recalls that k-itemsets generated in the second MapReduce job of FiDoop and FiDoop-HD are the input of the third MapReduce job (see Fig. 1). A small minimum support gives rise to an increasing number of k-itemsets to be decomposed by the third MapReduce job. For the Pfp case, the time spent in grouping and processing FP-tree goes up as the number of k-itemsets increases.

B. Load Balancing

In this group of experiments, we measure FiDoop's workload balance metric on a low- and high-dimensional datasets. In the case of high-dimensional dataset, we test our algorithms using the celestial spectral dataset. Recall that our analysis shows that the load balancing mechanism of the third MapReduce job substantially improves the performance of FiDoop. Section V-A formally introduces the workload balance metric. We measure FiDoop's workload balance metric when the celestial spectral dataset is used as an input.

We partition the initial input using the default settings. The workload balance metric defined in Section V-A does not incorporate the skewness of initial input; rather, the balance metric is measured based on the load of decomposing itemsets in the third MapReduce job. In this experiment, we use a partition function to obtain the measurement of workload balance metric [see (3)].

Fig. 4 shows the impact of workload balance metric on running time measured in the unit of 1000 s. The experimental results plotted in Fig. 4 clearly indicate that a Hadoop cluster with good intrinsic balance in workload leads to high performance of our FiDoop algorithm, because an increasing value of workload balance metric significantly shortens the running time of FiDoop and FiDoop-HD.

Importantly, the placement of initial input data is a major factor affecting the load-balance performance of FiDoop running on Hadoop clusters. Initial data placement has certain impacts on the performance of the first two MapReduce jobs in FiDoop. Because the third MapReduce job of FiDoop is more sensitive to the initial data placement than the first two MapReduce jobs, we are focusing on the load-balancing performance of the third MapReduce job rather the first two.

C. Speedup

We evaluate the speedup performance of Pfp, FiDoop, and FiDoop-HD by increasing the number of data nodes in the

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:32:48 UTC from IEEE Xplore. Restrictions apply.

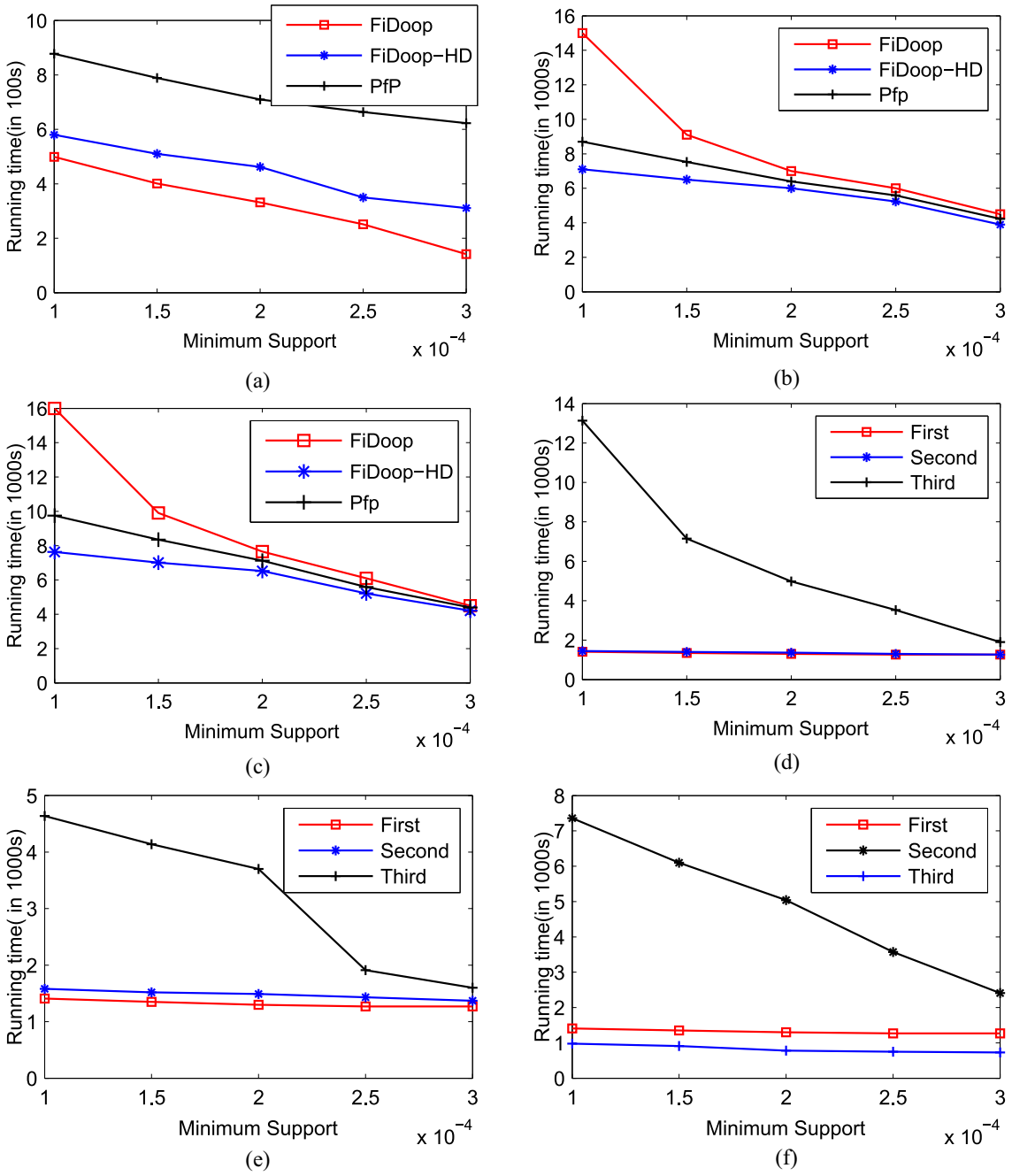


Fig. 3. Effect of minimum support on (a) 10 dimensions, (b) 40 dimensions, (c) celestial spectral dataset, (d) three stages of FiDoop, (e) three stages of FiDoop-HD, and (f) three stages of Pfp.

test Hadoop cluster from 4 to 16 with an increment of 2. The celestial spectral dataset is applied to drive the speedup analysis of the three algorithms. Fig. 5 reveals the speedup of the three schemes as a function of the number of data nodes in the Hadoop cluster.

The experimental results illustrated in Fig. 5(a) show that the speedups of the three algorithms scale linearly when the number of data nodes increases from 4 to 14. When the number of data nodes is further increased from 14 to 16, the speedup improvement marginally slows down. Such a speedup trend can be attributed to the fact that increasing the number of data nodes under a fixed input data size inevitably: 1) reduces the amount of itemsets being handled by each node and 2) increases communication overhead between mappers and reducers.

There is a slowdown in speedup improvement and the reason is twofold. First, each node has to load input itemsets from the HDFS; such input load has more noticeable impacts on FiDoop-HD than on FiDoop. Second, the cost of shuffling intermediate results between mappers and reducers goes up sharply when there are an excessive number of data nodes to store a fixed amount of input itemsets. Although the overall computing capacity is improved by increasing the number of nodes, the cost of synchronization and communication among data nodes tends to offset the gain in computing capacity. For example, the results plotted in Fig. 5(b) confirm that the

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:32:48 UTC from IEEE Xplore. Restrictions apply.

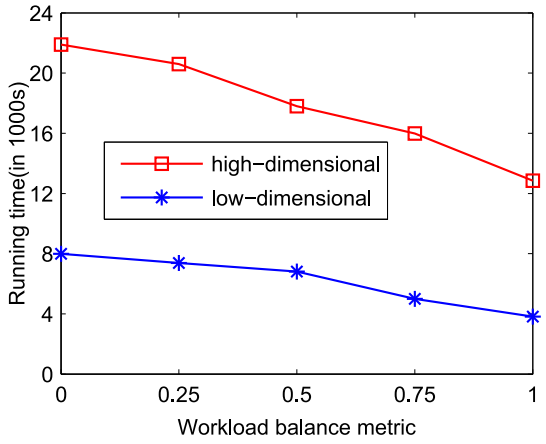
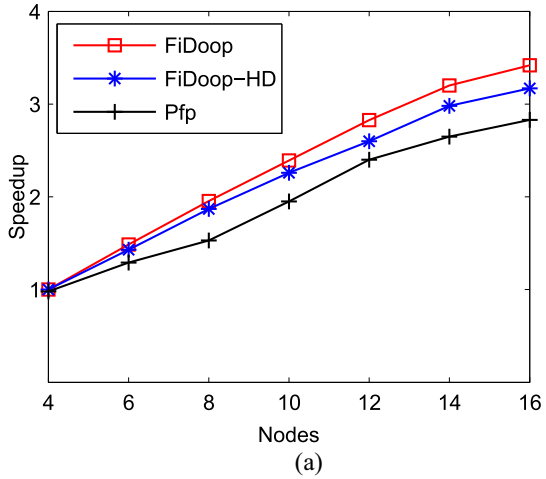
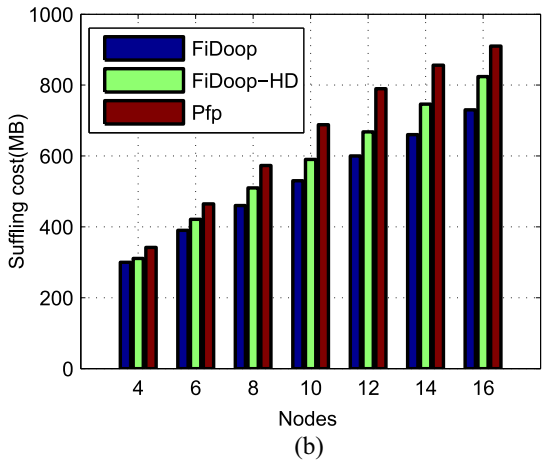


Fig. 4. Load-balancing performance of FiDoop.



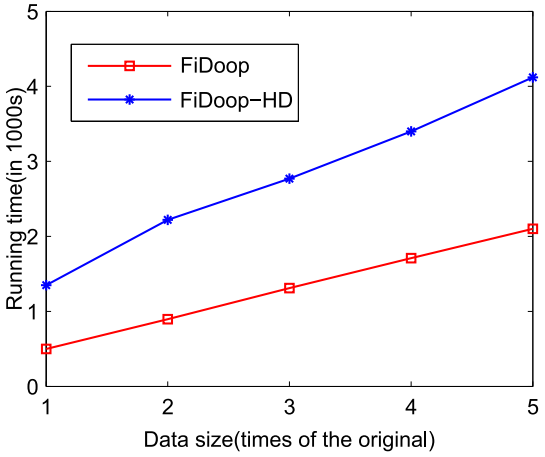
(a)



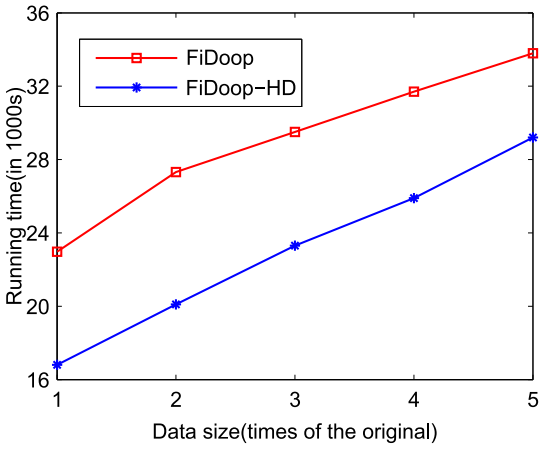
(b)

Fig. 5. (a) Speedup performance and (b) shuffling cost of FiDoop, FiDoop-HD, and Pfp when the number of data nodes increases from 4 to 16 with an increment of 2.

shuffling cost is linearly increasing when computing nodes are scaled from 4 to 16. Furthermore, the shuffling cost of Pfp is much larger than those of FiDoop and FiDoop-HD, because a large number of transactions have to be transmitted when they are not in the specified group of nodes. Consequently, Pfp is significantly inferior to FiDoop and FiDoop-HD in terms of the speedup performance.



(a)



(b)

Fig. 6. Scalability of FiDoop and FiDoop-HD when the size of input dataset increases. The number of dimensions is set to (a) 10 and (b) 40 dimensions.

D. Scalability

In this group of experiments, we evaluate the scalability of FiDoop when the size of input dataset grows dramatically. Fig. 6 shows the running time of FiDoop and FiDoop-HD when we scale up and process the dimensionality of the series of D1000W. Fig. 6(a) and (b) shows the performance of FiDoop and FiDoop-HD processing a low- and high-dimensional datasets, respectively.

Fig. 6 clearly reveals that the overall execution time of FiDoop and FiDoop-HD goes up when the input data size is sharply enlarged. The parallel mining process is slowed down by the excessive data amount that has to be scanned twice. The increased dataset leads to a long scanning time. Interestingly, FiDoop scales better than FiDoop-HD. For example, when data size is large, the cost of reading and writing files from and to HDFS in the FiDoop-HD case grows much faster than that of the FiDoop case.

Recall that (see Section VI) the outputs of the second MapReduce job are distributed and stored in intermediate files based on the length of itemset; these files are accessed by the third MapReduce job as an input. Further, the decomposed results are written into these external files. In summary, the scalability of FiDoop is higher than that of FiDoop-HD when it comes to parallel mining of an enormous amount of data.

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:32:48 UTC from IEEE Xplore. Restrictions apply.

Fig. 6(a) and (b) shows that when the dimension is relatively high, FiDoop-HD is superior to FiDoop in terms of execution time; FiDoop is faster than FiDoop-HD for datasets with low dimensions. This performance trend is expected, and the reason is twofold. First, the cost of decomposing itemsets in the third MapReduce job of FiDoop-HD is a whole lot smaller than that of FiDoop. Second, the performance improvement offered by FiDoop-HD during the third MapReduce job exceeds the FiDoop-HD's high cost of reading and writing intermediate files from and to HDFS. This performance trend becomes more pronounced for parallel mining of high-dimensional datasets.

The aforementioned experimental results conclude that for the case of large low-dimensional datasets, FiDoop and FiDoop-HD outperforms Pfp. More importantly, FiDoop-HD optimizes the performance of FiDoop for processing high-dimensional data (see Fig. 3); FiDoop-HD is superior to FiDoop when itemsets to be decomposed are large. Pfp is seemingly better than FiDoop when high-dimensional datasets are processed. Nevertheless, we observe from Fig. 5 that Pfp exhibits poor speedup and shuffling cost, because Pfp has to transmit a large number of transactions that are not in the specified group nodes. Compared with FiDoop and FiDoop-HD, Pfp has high space consumption due to recursively generating and searching conditional FP trees. Our findings suggest that one can judiciously choose a solution between FiDoop and FiDoop-HD depending on the dimensionality of input datasets.

VIII. Related Work

A. Mining of Frequent Itemsets

The *Apriori* algorithm is a classic way of mining frequent itemsets in a database [3]. A variety of *Apriori*-like algorithms aim to shorten database scanning time by reducing candidate itemsets. For example, Park *et al.* [20] proposed the direct hashing and pruning algorithm to control the number of candidate two-itemsets and prune the database size using a hash technique. In the inverted hashing and pruning algorithm [21], every k-itemset within each transaction is hashed into a hash table. Berzal *et al.* [22] designed the tree-based association rule algorithm, which employs an effective data-tree structure to store all itemsets to reduce the time required for scanning databases.

To improve the performance of *Apriori*-like algorithms, Han *et al.* [4] proposed a novel approach called FP-growth to avoid generating an excessive number of candidate itemsets. The main idea of FP-growth is projecting database into a compact data structure, and then using the divide-and-conquer method to extract frequent itemsets. The main bottlenecks of FP-growth are: 1) the construction of a large number of conditional FP trees residing in the main memory and 2) the recursive traverse of FP trees. To address this problem, Tsay *et al.* [13] proposed a new method called FIUT, which relies on frequent items ultrametric trees to avoid recursively traversing FP trees. Zhang *et al.* [23] proposed a concept of constrained frequent pattern trees to substantially improve the efficiency of mining association rules.

B. Parallel Mining of Frequent Itemsets

Parallel frequent itemsets mining algorithms based on *Apriori* can be classified into two camps, namely, count distribution (e.g., count distribution (CD) [6], fast parallel mining [24], and parallel data mining (PDM) [25]) and data distribution (e.g., data distribution (DD) [6] and intelligent data distribution [26]). In the count distribution camp, each processor of a parallel system calculates the local support counts of all candidate itemsets. Then, all processors compute the total support counts of the candidates by exchanging the local support counts. The CD and PDM algorithms have simple communication patterns, because in every iteration each processor requires only one round of communication. In the data distribution camp, each processor only keeps the support counts of a subset of all candidates. Each processor is responsible for sending its local database partition to all the other processors to compute support counts. In general, DD has higher communication overhead than CD, because shipping transaction data demands more communication bandwidth than sending support counts.

The cascade running mode in existing *Apriori*-based parallel mining algorithms leads to high communication and synchronization overheads. To reduce time required for scanning databases and exchanging candidate itemsets, FP-growth-based parallel algorithms were proposed as a replacement of the *Apriori*-based parallel algorithms. A few parallel FP-growth-based parallel algorithms (see [9], [10]) were implemented using multithreading on multicore processors. A major disadvantage of these parallel mining algorithms lies in the infeasibility to construct main-memory-based FP trees when databases are very large. This problem becomes pronounced when it comes to massive and multidimensional databases.

C. Parallel Data Mining on Clusters

Clusters and other shared-nothing multiprocessor systems are scalable computing platforms that address the aforementioned main memory issue raised in parallel mining of large-scale databases. For example, Pramudiono and Kitsuregawa [27] proposed a parallel FP-growth algorithm running on a cluster. Javed and Khokhar [11] developed an efficient parallel algorithm using message passing interface on a shared-nothing multiprocessor system. Their PFP-tree-based parallel algorithm minimizes synchronization overheads by efficiently partitioning FP-tree and the frequent-element list over processors. Tang and Turkia [28] used the extended conditional databases and k-prefix search space partitioning to parallelize FIM, and an implementation of the new scheme with FP trees is presented. Yu and Zhou [29] proposed two parallel mining algorithms, Tidset-based parallel FP-tree (TPFP-tree) and balanced Tidset-based parallel FP-tree (BTP-tree). The TPFP-tree algorithm uses a transaction identification set to directly select transactions rather than scanning an entire database with a vertical database layout. Like most parallel mining algorithms running on clusters, TPFP-tree was implemented using the message passing interface programming model.

Load balancing techniques have been exploited to improve performance of parallel mining algorithms. Cong *et al.* [30]

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:32:48 UTC from IEEE Xplore. Restrictions apply.

designed a sampling-based framework for parallel data mining. This framework addressed the load balancing issues by partitioning frequent items and assigning each frequent-item subset to a processor. After computing the projection of databases for assigned items, each processor asynchronously mines its projected database in a pattern-growth manner without interprocessor communication. To optimize the performance of heterogeneous computing environments, Yu *et al.* [31] developed the BTP-tree algorithm, which is an extension of the TPFP-tree algorithm. The load balancing scheme adopted in the BTP-tree algorithm takes into account the heterogeneous processing power of computing nodes, thereby delivering an effective approach to parallel mining of frequent patterns. Zhou *et al.* [32] proposed a balanced parallel FP-growth algorithm balanced parallel FP-growth based on the Pfp algorithm [18].

Recently, increasing attention has been paid to supporting parallel data mining over cloud computing environments and the MapReduce framework. For example, Li *et al.* [18] proposed the Pfp algorithm on distributed machines. Pfp partitions computation to make each machine execute an independent group of mining tasks, thus, eliminating communication cost. Lin *et al.* [33] proposed a method for mining frequent itemsets from very large database. It used an efficient data structure for storing and retrieving FP trees and used the disk as the secondary memory. Yang *et al.* [34] proposed a distributed Distributed Hash-Trie frequent pattern algorithm using java persistence API based on Hadoop. Riondato *et al.* [35] proposed PARMA which cut down the data-size-dependent part of the cost by using a random sampling approach to FIM to improve the efficiency of mining. Hong *et al.* [36] proposed an improved FP-growth algorithm in MapReduce for discovering frequent patterns. Choi *et al.* [37] designed a scheme for applying MapReduce to the FP-growth algorithm to ensure efficient distribution of important resources (CPU, network, and storage). Different from the above techniques that make use of FP trees, our FiDoop incorporates FIU trees to achieve compressed storage and avoid building conditional pattern bases.

IX. Conclusion

To solve the scalability and load balancing challenges in the existing parallel mining algorithms for frequent itemsets, we applied the MapReduce programming model to develop a parallel frequent itemsets mining algorithm called FiDoop. FiDoop incorporates the frequent items ultrametric tree or FIU-tree rather than conventional FP trees, thereby achieving compressed storage and avoiding the necessity to build conditional pattern bases. FiDoop seamlessly integrates three MapReduce jobs to accomplish parallel mining of frequent itemsets. The third MapReduce job plays an important role in parallel mining; its mappers independently decompose itemsets whereas its reducers construct small ultrametric trees to be separately mined. We improve the performance of FiDoop by balancing I/O load across data nodes of a cluster.

We designed and implemented FiDoop-HD—an extension of FiDoop—to efficiently handle high-dimensional data processing. FiDoop-HD decomposes the M-itemsets into a list of $(M-1)$-itemsets, which are further decomposed into $(M-2)$-itemsets to be unioned into the original $(M-2)$-itemsets. This procedure is repeatedly carried out until the entire decomposition process is accomplished.

We implemented FiDoop and FiDoop-HD on a real-world Hadoop cluster. The extensive experiments using real-world Celestial Spectral data show that our proposed solution is sensitive to both data distribution and dimensions, because the lengths of itemsets have significant impacts on decomposition and construction cost. The experimental results reveal that for large low-dimensional datasets, FiDoop performs better than FiDoop-HD, where FiDoop-HD outperforms FiDoop for high-dimensional data processing. FiDoop-HD improves the performance of FiDoop when long itemsets are decomposed. Our new findings suggest that it is flexible for data mining users to judiciously switch a solution between FiDoop and FiDoop-HD depending on the size of input itemsets and the number of dimensions.

In this paper, we introduced a metric to measure the load balance of FiDoop. As a future research direction, we will apply this metric to investigate advanced load balance strategies in the context of FiDoop. For example, we plan to implement a data-aware load balancing scheme to substantially improve the load-balancing performance of FiDoop. In one of our previous studies (see [38]), we have addressed the data-placement issue in heterogeneous Hadoop clusters, where data are placed across nodes in a way that each node has a balanced data processing load. Our data placement scheme [38] is conducive to balancing the amount of data stored in each heterogeneous node to achieve improved data-processing performance. We will integrate FiDoop with the data-placement mechanism on heterogeneous clusters. One of the goals is to investigate the impact of heterogeneous data placement strategy on Hadoop-based parallel mining of frequent itemsets. In addition to performance issues, energy savings and thermal management will be of our future research interests. We will propose various approaches to improving energy efficiency of Fidoop running on Hadoop clusters.

Acknowledgment

The authors would like to thank M. Lau for proofreading.

References

[1] M. J. Zaki, "Parallel and distributed association mining: A survey," *IEEE Concurrency*, vol. 7, no. 4, pp. 14–25, Oct./Dec. 1999.

[2] I. Pramudiono and M. Kitsuregawa, "FP-tax: Tree structure based generalized association rule mining," in *Proc. 9th ACM SIGMOD Workshop Res. Issues Data Min. Knowl. Disc.*, Paris, France, 2004, pp. 60–63.

[3] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," *ACM SIGMOD Rec.*, vol. 22, no. 2, pp. 207–216, 1993.

[4] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining frequent patterns without candidate generation: A frequent-pattern tree approach," *Data Min. Knowl. Disc.*, vol. 8, no. 1, pp. 53–87, 2004.

[5] S. Kotsiantis and D. Kanellopoulos, "Association rules mining: A recent overview," *GESTS Int. Trans. Comput. Sci. Eng.*, vol. 32, no. 1, pp. 71–82, 2006.

[6] R. Agrawal and J. C. Shafer, "Parallel mining of association rules," *IEEE Trans. Knowl. Data Eng.*, vol. 8, no. 6, pp. 962–969, Dec. 1996.

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:32:48 UTC from IEEE Xplore. Restrictions apply.

[7] A. Schuster and R. Wolff, "Communication-efficient distributed mining of association rules," *Data Min. Knowl. Disc.*, vol. 8, no. 2, pp. 171–196, 2004.
[8] M.-Y. Lin, P.-Y. Lee, and S.-C. Hsueh, "Apriori-based frequent itemset mining algorithms on MapReduce," in *Proc. 6th Int. Conf. Ubiquit. Inf. Manage. Commun. (ICUIMC)*, Danang, Vietnam, 2012, pp. 76:1–76:8. [Online]. Available: http://doi.acm.org/10.1145/2184751.2184842
[9] D. Chen *et al.*, "Tree partition based parallel frequent pattern mining on shared memory systems," in *Proc. 20th IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, Rhodes Island, Greece, 2006, pp. 1–8.
[10] L. Liu, E. Li, Y. Zhang, and Z. Tang, "Optimization of frequent itemset mining on multiple-core processor," in *Proc. 33rd Int. Conf. Very Large Data Bases*, Vienna, Austria, 2007, pp. 1275–1285.
[11] A. Javed and A. Khokhar, "Frequent pattern mining on message passing multiprocessor systems," *Distrib. Parallel Databases*, vol. 16, no. 3, pp. 321–334, 2004.
[12] J. Neerbek, "Message-driven FP-growth," in *Proc. WICSA/ECSA Compan. Vol.*, Helsinki, Finland, 2012, pp. 29–36.
[13] Y.-J. Tsay, T.-J. Hsu, and J.-R. Yu, "FIUT: A new method for mining frequent itemsets," *Inf. Sci.*, vol. 179, no. 11, pp. 1724–1737, 2009.
[14] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.
[15] J. Dean and S. Ghemawat, "MapReduce: A flexible data processing tool," *Commun. ACM*, vol. 53, no. 1, pp. 72–77, Jan. 2010.
[16] W. Lu, Y. Shen, S. Chen, and B. C. Ooi, "Efficient processing of k nearest neighbor joins using MapReduce," *Proc. VLDB Endow.*, vol. 5, no. 10, pp. 1016–1027, 2012.
[17] D. W. Cheung, S. D. Lee, and Y. Xiao, "Effect of data skewness and workload balance in parallel data mining," *IEEE Trans. Knowl. Data Eng.*, vol. 14, no. 3, pp. 498–514, May/Jun. 2002.
[18] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang, "PFP: Parallel FP-growth for query recommendation," in *Proc. ACM Conf. Recommend. Syst.*, Lausanne, Switzerland, 2008, pp. 107–114.
[19] L. Cristofor. (2001). *Artool Project[J]*. [Online]. Available: http://www.cs.umb.edu/laur/ARtool/, accessed Oct. 19, 2012.
[20] J. S. Park, M.-S. Chen, and P. S. Yu, "Using a hash-based method with transaction trimming for mining association rules," *IEEE Trans. Knowl. Data Eng.*, vol. 9, no. 5, pp. 813–825, Sep./Oct. 1997.
[21] J. D. Holt and S. M. Chung, "Mining association rules using inverted hashing and pruning," *Inf. Process. Lett.*, vol. 83, no. 4, pp. 211–220, 2002.
[22] F. Berzal, J.-C. Cubero, N. Marín, and J.-M. Serrano, "TBAR: An efficient method for association rule mining in relational databases," *Data Knowl. Eng.*, vol. 37, no. 1, pp. 47–64, 2001.
[23] J. Zhang, X. Zhao, S. Zhang, S. Yin, and X. Qin, "Interrelation analysis of celestial spectra data using constrained frequent pattern trees," *Knowl.-Based Syst.*, vol. 41, pp. 77–88, Mar. 2013.
[24] D. W. Cheung and Y. Xiao, "Effect of data skewness in parallel mining of association rules," in *Research and Development in Knowledge Discovery and Data Mining*. Berlin, Germany: Springer, 1998, pp. 48–60.
[25] T. Shintani and M. Kitsuregawa, "Hash based parallel algorithms for mining association rules," in *Proc. 4th Int. Conf. Parallel Distrib. Inf. Syst.*, Miami Beach, FL, USA, 1996, pp. 19–30.
[26] E.-H. Han, G. Karypis, and V. Kumar, "Scalable parallel data mining for association rules," *IEEE Trans. Knowl. Data Eng.*, vol. 12, no. 3, pp. 337–352, May/Jun. 2000.
[27] I. Pramudiono and M. Kitsuregawa, "Parallel FP-growth on PC cluster," in *Advances in Knowledge Discovery and Data Mining*. Berlin, Germany: Springer, 2003, pp. 467–473.
[28] P. Tang and M. P. Turkia, "Parallelizing frequent itemset mining with FP-trees," in *Proc. 21st Int. Conf. Comput. Appl.*, Seattle, WA, USA, 2006, pp. 30–35.
[29] K. Yu and J. Zhou, "Parallel TID-based frequent pattern mining algorithm on a PC cluster and grid computing system," *Expert Syst. Appl.*, vol. 37, no. 3, pp. 2486–2494, 2010.
[30] S. Cong, J. Han, J. Hoeflinger, and D. Padua, "A sampling-based framework for parallel data mining," in *Proc. 10th ACM SIGPLAN Symp. Prin. Pract. Parallel Program.*, Chicago, IL, USA, 2005, pp. 255–265.
[31] K.-M. Yu, J. Zhou, T.-P. Hong, and J.-L. Zhou, "A load-balanced distributed parallel mining algorithm," *Expert Syst. Appl.*, vol. 37, no. 3, pp. 2459–2464, 2010.
[32] L. Zhou *et al.*, "Balanced parallel FP-growth with MapReduce," in *Proc. IEEE Youth Conf. Inf. Comput. Telecommun. (YC-ICT)*, Beijing, China, 2010, pp. 243–246.
[33] K. W. Lin, P.-L. Chen, and W.-L. Chang, "A novel frequent pattern mining algorithm for very large databases in cloud computing environments," in *Proc. IEEE Int. Conf. Granular Comput. (GrC)*, Kaohsiung, Taiwan, 2011, pp. 399–403.
[34] L. Yang, Z. Shi, L. D. Xu, F. Liang, and I. Kirsh, "DH-TRIE frequent pattern mining on Hadoop using JPA," in *Proc. IEEE Int. Conf. Granular Comput. (GrC)*, Kaohsiung, Taiwan, 2011, pp. 875–878.
[35] M. Riondato, J. A. DeBrabant, R. Fonseca, and E. Upfal, "PARMA: A parallel randomized algorithm for approximate association rules mining in MapReduce," in *Proc. 21st ACM Int. Conf. Inf. Knowl. Manage.*, Maui, HI, USA, 2012, pp. 85–94.
[36] S. Hong, Z. Huaxuan, C. Shiping, and H. Chunyan, "The study of improved FP-growth algorithm in MapReduce," in *Proc. 1st Int. Workshop Cloud Comput. Inf. Security*, Shanghai, China, 2013, pp. 250–253.
[37] J. Choi, C. Choi, K. Yim, J. Kim, and P. Kim, "Intelligent reconfigurable method of cloud computing resources for multimedia data delivery," *Informatica*, vol. 24, no. 3, pp. 381–394, 2013.
[38] J. Xie and X. Qin, "The 19th heterogeneity in computing workshop (HCW 2010)," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Workshops Ph.D. Forum (IPDPSW)*, Atlanta, GA, USA, Apr. 2010, pp. 1–5.



Yaling Xun is currently pursuing the doctoral degree with the Taiyuan University of Science and Technology (TYUST), Taiyuan, China.

She is a Lecturer with the School of Computer Science and Technology, TYUST. Her current research interests include data mining and parallel computing.



Jifu Zhang received the B.S. and M.S. degrees in computer science and technology from the Hefei University of Technology, Hefei, China, in 1983 and 1989, respectively, and the Ph.D. degree in pattern recognition and intelligence systems from the Beijing Institute of Technology, Beijing, China, in 2005.

He is currently a Professor with the School of Computer Science and Technology, Taiyuan University of Science and Technology, Taiyuan, China. His current research interests include data mining, parallel and distributed computing, and artificial intelligence.



Xiao Qin (SM'09) received the B.S. and M.S. degrees from the Huazhong University of Science and Technology, Wuhan, China, in 1992 and 1999, respectively, and the Ph.D. degree from the University of Nebraska-Lincoln, Lincoln, NE, USA, in 2004, all in computer science.

He is currently an Associate Professor with the Department of Computer Science and Software Engineering, Auburn University, Auburn, AL, USA. His current research interests include parallel and distributed systems, storage systems, fault tolerance, real-time systems, and performance evaluation.

Dr. Qin was a recipient of the U.S. National Sanitation Foundation (NSF) Computing Processes and Artifacts Award and the NSF Computer System Research Award in 2007 and the NSF CAREER Award in 2009.

Authorized licensed use limited to: Auburn University. Downloaded on November 18,2026 at 18:32:48 UTC from IEEE Xplore. Restrictions apply.