This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2560176, IEEE Transactions on Parallel and Distributed Systems

FiDoop-DP: Data Partitioning in Frequent Itemset Mining on Hadoop Clusters

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

Abstract—Traditional parallel algorithms for mining frequent itemsets aim to balance load by equally partitioning data among a group of computing nodes. We start this study by discovering a serious performance problem of the existing parallel Frequent Itemset Mining algorithms. Given a large dataset, data partitioning strategies in the existing solutions suffer high communication and mining overhead induced by redundant transactions transmitted among computing nodes. We address this problem by developing a data partitioning approach called FiDoop-DP using the MapReduce programming model. The overarching goal of FiDoop-DP is to boost the performance of parallel Frequent Itemset Mining on Hadoop clusters. At the heart of FiDoop-DP is the Voronoi diagram-based data partitioning technique, which exploits correlations among transactions. Incorporating the similarity metric and the Locality-Sensitive Hashing technique, FiDoop-DP places highly similar transactions into a data partition to improve locality without creating an excessive number of redundant transactions. We implement FiDoop-DP on a 24-node Hadoop cluster, driven by a wide range of datasets created by IBM Quest Market-Basket Synthetic Data Generator. Experimental results reveal that FiDoop-DP is conducive to reducing network and computing loads by the virtue of eliminating redundant transactions on Hadoop nodes. FiDoop-DP significantly improves the performance of the existing parallel frequent-pattern scheme by up to 31% with an average of 18%.

Index Terms—Frequent Itemset Mining, Parallel Data Mining, Data Partitioning, MapReduce Programming Model, Hadoop Cluster

1 Introduction

Traditional parallel Frequent Itemset Mining techniques (a.k.a., FIM) are focused on load balancing; data are equally partitioned and distributed among computing nodes of a cluster. More often than not, the lack of analysis of correlation among data leads to poor data locality. The absence of data collocation increases the data shuffling costs and the network overhead, reducing the effectiveness of data partitioning. In this study, we show that redundant transaction transmission and itemset-mining tasks are likely to be created by inappropriate data partitioning decisions. As a result, data partitioning in FIM affects not only network traffic but also computing loads. Our evidence shows that data partitioning algorithms should pay attention to network and computing loads in addition to the issue of load balancing. We propose a parallel FIM approach called FiDoop-DP using the MapReduce programming model. The key idea of FiDoop-DP is to group highly relevant transactions into a data partition; thus, the number of redundant transactions is significantly slashed. Importantly, we show how to partition and distribute a large dataset across data nodes of a Hadoop cluster to reduce network and computing loads induced by making redundant transactions on remote nodes. FiDoop-DP is conducive to speeding up the performance of parallel FIM on clusters.

- *Y. Xun, J. Zhang* and X. Zhao, are with Taiyuan University of Science and Technology (TYUST), Taiyuan, Shanxi, China. 030024. E-Mail: zjf@tyust.edu.cn, *corresponding author: zjf@tyust.edu.cn.*
- *X. Qin is with the Department of Computer Science and Software Engineering, Samuel Ginn College of Engineering, Auburn University, AL 36849-5347. E-mail: xqin@auburn.edu.*

1.1 Motivations

The following three observations motivate us to develop FiDoop-DP in this study to improve the performance of FIM on high-performance clusters.

- There is a pressing need for the development of parallel FIM techniques.
- The MapReduce programming model is an ideal data-centric mode to address the rapid growth of big-data mining.
- Data partitioning in Hadoop clusters play a critical role in optimizing the performance of applications processing large datasets.

Parallel Frequent Itemset Mining. Datasets in modern data mining applications become excessively large; therefore, improving performance of FIM is a practical way of significantly shortening data mining time of the applications. Unfortunately, sequential FIM algorithms running on a single machine suffer from performance deterioration due to limited computational and storage resources [1][2]. To fill the deep gap between massive amounts of datasets and sequential FIM schemes, we are focusing on parallel FIM algorithms running on clusters.

The MapReduce Programming Model. MapReduce - a highly scalable and fault-tolerant parallel programming model - facilitates a framework for processing large scale datasets by exploiting parallelisms among data nodes of a cluster [3][4]. In the realm of big data processing, MapReduce has been adopted to develop parallel data mining algorithms, including Frequent Itemset Mining (e.g., Apriori-based [5][6], FP-Growth-based [7][8], as well as other classic association rule mining [9]). Hadoop is an open source im-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2560176, IEEE Transactions on Parallel and Distributed Systems

plementation of the MapReduce programming model [10]. In this study, we show that Hadoop cluster is an ideal computing framework for mining frequent itemsets over massive and distributed datasets.

Data Partitioning in Hadoop Clusters. In modern distributed systems, execution parallelism is controlled through data partitioning which in turn provides the means necessary to achieve high efficiency and good scalability of distributed execution in a large-scale cluster. Thus, efficient performance of data-parallel computing heavily depends on the effectiveness of data partitioning. Existing data partitioning solutions of FIM built in Hadoop aim at balancing computation load by equally distributing data among nodes. However, the correlation between the data is often ignored which will lead to poor data locality, and the data shuffling costs and the network overhead will increase. We develop FiDoop-DP, a parallel FIM technique, in which a large dataset is partitioned across a Hadoop cluster's data nodes in a way to improve data locality.

1.2 Data Partitioning Problems Solved in FiDoop-DP

In Hadoop clusters, the amount of transferred data during the shuffling phase heavily depends on localities and balance of intermediate results. Unfortunately, when a data partitioning scheme partitions the intermediate results, data locality and balance are completely ignored. In the existing Hadoop-based FIM applications [7][8][11], the traditional data partitioning schemes impose a major performance problem due to the following reasons:

Conventional wisdoms in data partitioning aim to yield balanced partitions using either a hash function or a set of equally spaced range keys [12][13]. Interestingly, we discover that excessive computation and network loads are likely to be caused by inappropriate data partitions in parallel FIM. Fig. 1 offers a motivational example showing various item grouping and data partitioning decisions and their effects on communication and computing load. In Fig. 1, each row in the middle table represents a transaction (i.e., a total of ten transactions); twelve items (e.g., f, c, a, etc.) are managed in the transaction database (see the left-hand and right-hand columns in Fig. 1). Note that the two columns indicate two grouping strategies divided by a midline. The traditional grouping strategy evenly groups the items into two groups by descending frequency (see the column on the left-hand side of Fig. 1). Unfortunately, this grouping decision forces all the transactions to be transmitted to the two partitions prior to being processed. We argue that such a high transaction-transfer overhead can be reduced by making a good tradeoff between cross-node network traffic and load balancing.

In a multi-stage parallel process of mining frequent itemsets, redundant mining tasks tend to occur in later stages. It is more often than not difficult to predict such redundant tasks before launching the parallel mining program. Hence, existing data partitioning algorithms that performed prior to the parallel mining process are inadequate for solving the problem of redundant tasks.

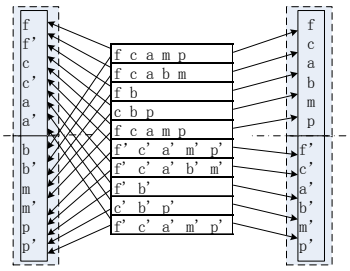


Fig. 1. A motivational example of items grouping and data partitioning.

1.3 Basic Ideas

The overarching goal of FiDoop-DP is to boost the performance of parallel FIM applications running on Hadoop clusters. This goal is achieved in FiDoop-DP by reducing network and computing loads through the elimination of redundant transactions on multiple nodes. To alleviate the excessive network load problem illustrated in Fig. 1, we show that discovering correlations among items and transactions create ample opportunities to significantly reduce the transaction transfer overhead (see the column on the right-hand side of Fig. 1). This new grouping decision makes it possible to construct small FP trees, which in turn lower communication and computation cost.

We incorporate the data partitioning scheme into Hadoop-based frequent-pattern-tree (FP-tree) algorithms. In addition to FP-tree algorithms (e.g., FP-Growth [14] and FUIT [15]), other FIM algorithms like Apriori [5][6] can benefit from our data partitioning idea (see further discussions in Section 8). Fig. 2 outlines the typical process flow (see also [11]) adopted by our FiDoop-DP, which consists of four steps. In this process flow, we optimize the data partitioning strategy of the second MapReduce job, because it is the most complicated and time-consuming job in FiDoop-DP. In the second MapReduce job, the mappers divide frequent 1-itemsets (FList in Fig.2) into Q groups, while simultaneously assigning transactions to computing nodes based on the grouping information. Then, the reducers concurrently perform mining tasks for the partitioned groups.

In the mappers of the second MapReduce job, we propose a novel way of incorporating LSH (a.k.a., Locality Sensitive Hashing) scheme into Voronoi diagram-based partitioning, thereby clustering similar transactions together and determining correlation degrees among the transactions. Next, frequent items produced by the first MapReduce job are grouped according to the correlation degrees among items, and transactions are partitioned. This frequent-items grouping and partitioning strategy is capable of reducing the number of redundant transactions kept on multiple nodes and, as a result, both data transmission traffic and redundant computing load are significantly decreased.

1.4 Contributions

We summarize the main contributions of this study as follows:

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2560176, IEEE Transactions on Parallel and Distributed Systems

- In the context of FIM, we design an efficient data partitioning scheme, which facilitates an analysis of correlations among transactions to reduce network and computing load. Our scheme prevents transactions from being repeatedly transmitted across multiple nodes.
- We implement the above data partitioning scheme by integrating Voronoi-diagram with LSH (Locality-Sensitive Hashing).
- To validate the effectiveness of our approach, we develop the FiDoop-DP prototype, where the data partitioning scheme is applied to a Hadoop-based FP-Growth algorithm.
- We conduct extensive experiments using synthetic datasets to show that FiDoop-DP is robust, efficient, and scalable on Hadoop clusters.

1.5 Roadmap

The remainder of this paper is organized as follows. Section 2 describes the background knowledge. Section 3 summarizes the traditional solutions and formulates the data partitioning problem. Section 4 presents the design issues of FiDoop-DP built on the MapReduce framework, followed by the implementation details in Section 5. Section 6 evaluates the performance of FiDoop-DP on a real-world cluster. Section 7 discusses the related work. Finally, Section 8 and 9 conclude the paper with future research directions.

2 PRELIMINARIES

In this section, we first briefly review FIM. Then, to facilitate the presentation of FiDoop-DP, we introduce the MapReduce programming framework. Finally, we summarize the basic idea of Parallel FP-Growth Algorithm - Pfp [11] which has been implemented in mahout [16]. We use Pfp as a case study to demonstrate that data partitioning can help in improving the performance of FIM.

2.1 Frequent Itemset Mining

Frequent Itemset Mining is one of the most critical and time-consuming tasks in association rule mining (ARM), an often-used data mining task, provides a strategic resource for decision support by extracting the most important frequent patterns that simultaneously occur in a large transaction database. A typical application of ARM is the famous market basket analysis.

In FIM, support is a measure defined by users. An itemset X has support s if s% of transactions contain the itemset. We denote $s = support(X)$; the support of the rule $X \Rightarrow Y$ is support($X \cup Y$). Here X and Y are two itemsets, and $X \cap Y = \emptyset$. The purpose of FIM is to identify all frequent itemsets whose support is greater than the minimum support. The first phase is more challenging and complicated than the second one. Most prior studies are primarily focused on the issue of discovering frequent itemsets.

2.2 MapReduce Framework

MapReduce is a popular data processing paradigm for efficient and fault tolerant workload distribution in large clusters. A MapReduce computation has two phases, namely, the Map phase and the Reduce phase. The Map phase splits an input data into a large number of fragments, which are evenly distributed to Map tasks across a cluster of nodes 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. MapReduce applies the main idea of moving computation towards data, scheduling map tasks to the closest nodes where the input data is stored in order to maximize data locality.

Hadoop is one of the most popular MapReduce implementations. Both input and output pairs of a MapReduce application are managed by an underlying Hadoop distributed file system (HDFS [17]). At the heart of HDFS is a single NameNode a master server managing the file system namespace and regulates file accesses. The Hadoop runtime system establishes two processes called JobTracker and TaskTracker. Job-Tracker is responsible for assigning and scheduling tasks; each TaskTracker handles mappers or reducers assigned by JobTracker.

When Hadoop exhibits an overwhelming development momentum, a new MapReduce programming model Spark attracts researchers' attention [18]. The main abstraction in Spark is a resilient distributed dataset (RDD), which offers good fault tolerance and allows jobs to perform computations in memory on large clusters. Thus, Spark becomes an attractive programming model to iterative MapReduce algorithms. We decide to develop FiDoop-DP on Hadoop clusters; in a future study, we plan to extend FiDoop-DP to Spark to gain further performance improvement.

2.3 Parallel FP-Growth Algorithm

In this study, we focus on a popular FP-Growth algorithm called Parallel FP-Growth or Pfp for short [11]. Pfp implemented in Mahout [16] is a parallel version of the FP-Growth algorithm [2]. Mahout is an open source machine learning library developed on Hadoop clusters. FP-Growth efficiently discovers frequent itemsets by constructing and mining a compressed data structure (i.e., FP-tree) rather than an entire database. Pfp was designed to address the synchronization issues by partitioning transaction database into independent partitions, because it is guaranteed that each partition contains all the data relevant to the features (or items) of that group.

Given a transaction database DB, Fig.2 depicts the process flow of Parallel FP-Growth implemented in Mahout. The parallel algorithm consists of four steps, three of which are MapReduce jobs.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2560176, IEEE Transactions on Parallel and Distributed Systems

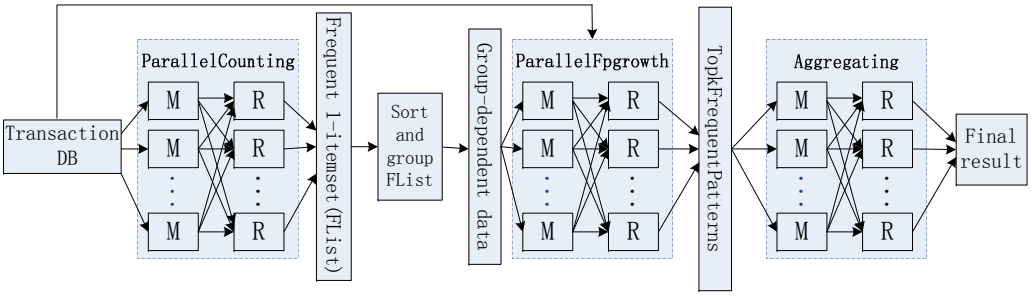


Fig. 2. The process flow of Pfp implemented in Mahout.

Step 1. Parallel Counting: The first MapReduce job counts the support values of all items residing in the database to discover all frequent items or frequent 1-itemsets in parallel. It is worth noting that this step simply scans the database once.

Step 2. Sorting frequent 1-itemsets to $FList$: The second step sorts these frequent 1-itemsets in a decreasing order of frequency; the sorted frequent 1-itemsets are cached in a list named $FList$. Step 2 is a non-MapReduce process due to its simplicity as well as the centralized control.

Step 3. Parallel FP-Growth: This is a core step of Pfp, where the map stage and reduce stage perform the following two important functions.

- *Mapper - Grouping items and generating group-dependent transactions.* First, the Mappers divide all the items in $FList$ into Q groups. The list of groups is referred to as group list or $GList$, where each group is assigned a unique group ID (i.e., Gid). Then, the transactions are partitioned into multiple groups according to GLists. That is, each mapper outputs one or more key-value pairs, where a keys is a group ID and its corresponding value is a generated group-dependent transaction.
- *Reducer - FP-Growth on group-dependent partitions.* local FPGrowth is conducted to generate local frequent itemsets. Each reducer conducts local FPGrowth by processing one or more group-dependent partition one by one, and discovered patterns are output in the final.

Step 4. Aggregating: The last MapReduce job produces final results by aggregating the output generated in Step 3.

The second MapReduce job (i.e., Step 3) is a performance bottleneck of the entire data mining process. The map tasks apply a second-round scan to sort and prune each transaction according to $FList$, followed by grouping the sorted frequent 1-itemsets in $FList$ to form group list $GList$. Next, each transaction is placed into a group-dependent data partition; thus, multiple data partitions are constructed. Each data partition corresponds to a group identified by Gid.

The above partitioning approach ensures data completeness with respect to one group of $GList$. A downside is that such data completeness comes at the cost of data redundancy, because a transaction might have duplicated copies in multiple data partitions. Not surprisingly, the data redundancy in data partitions are inevitable, because independence among the partitions has to be maintained to minimize synchronization overhead. Redundant transactions incur excessive data transfer cost and computing load of local FP-Growth.

3 PROBLEM STATEMENT

3.1 Baseline Methods and Problems

Evidence [7] shows that most existing parallel FP-Growth algorithms basically followed the workflow plotted in Fig. 2, where the second MapReduce job is the most performance critical and time-consuming among the four steps. Experiment results reported in [7] suggest that (1) local FP-Growth cost accounts for more than 50% of the overall mining time and (2) the grouping strategy plays the most important role in affecting subsequent data partitioning and local FP-Growth performance.

Reordered transactions are partitioned and assigned to corresponding reducers, each of which inserts the transactions into an FP-tree using the grouping strategy. That is, the grouping strategy not only directly governs the amount of transferred data in the partitioning stage, but also affects computing load of the local FP-Growth stage. To alleviate the problem of expensive grouping, we propose to cluster input data prior to running the grouping and partitioning stages. Our input data partitioning policy takes into account the correlations among transactions to optimize the grouping process.

A straightforward MapReduce-based FIM method is to adopt the default data partitioning policy implemented in Hadoop; then, a simple grouping strategy (see [11]) is applied. The grouping strategy first computes group size, which equals to the total number of frequent 1-itemsets in $FList$ divided by number of groups.

Let $GList_i$ be a set of items that belong to the ith group of $GList$. One can easily determine what items should be included in set $GList_i$ $(i > 0)$ by evenly distributing all the items into the groups. Specifically, the first item in $GList_i$ is the jth item in $FList$; j is calculated as $(\sum_{i=0}^{i-1} \mid GList_i \mid)+1$. Shuffling cost and computing load are not intentionally reduced in existing parallel FIM algorithms such as the Pfp algorithm implemented in Mahout.

An improvement to the aforementioned grouping and partitioning strategy is to incorporate a load balancing

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2560176, IEEE Transactions on Parallel and Distributed Systems

feature in Pfp (see, for example, the balanced parallel FP-Growth algorithm or BPFP [7]). BPFP divides all the items in $FList$ into Q groups in a way to balance load among computing nodes during the entire mining process. BPFP estimates mining load using the number of recursive iterations during the course of FP-Growth execution, the input of which is conditional pattern bases of each item. The location of each item in $FList$ is estimated to be the length of the longest path in the conditional pattern base. Meanwhile, the number of recursive iterations is exponentially proportional to the longest path in the conditional pattern base. Thus, the load of item i can be estimated as $T_i = logL_i$, where T_i represents the estimated load and L_i represents the location of item i in $FList$. As can be seen from the aforementioned description, BPFP only concerns the balance of CPU resource for each node by evenly dividing all computing load among the Q groups. However, fig. 2 shows when one partitions items into grouped without considering the correlation among transactions, an excessive number of duplicated transactions must be transmitted among the nodes in order to guarantee data completeness with respect to each group. In other words, the number of transferred transactions coupled with participating computing inevitably increases; thus, data transfer overhead (i.e., shuffling cost) and FIM load tend to be significant.

3.2 Design Goals

FiDoop-DP aims to partition input transactions (1) to reduce the amount of data transferred through the network during the shuffle phase and (2) to minimize local mining load. Recall that high shuffling cost and local mining load are incurred by redundant transactions. In what follows, we formally state the design goal of Fidoop-DP.

Let the input data for a MapReduce job be a set of transactions $D = \{t_1, t_2, ..., t_n\}$ and function $DBPart : D \rightarrow C$ partitions D into a set of chunks $C = \{C_1, C_2, ..., C_p\}$. Correspondingly, map tasks $M = \{m_1, m_2..., m_p\}$ and reduce tasks $R = \{r_1, r_2..., r_q\}$ are running on a cluster. We denote a set of intermediate key-value pairs produced by the mappers as $I = \{(G_1, D_1), ..., (G_m, D_m)$, in which D_i represents the collection of transactions belonging to group G_i. Intuitively, we have $output(m_i) \subseteq I$ and $input(r_i) \subseteq I$, where $output(m_i)$ and $input(r_i)$ respectively represent a set of intermediate pairs produced by map task m_i and a set of intermediate pairs assigned to reduce task r_i. After Map tasks are completed, the shuffle phase applies the default partitioning function to assign intermediate key-value pairs to reduce tasks according to the keys (i.e., G_i) of $output(m_i)$. In this process, if intermediate key-value pair $((G_i, D_i))$ is partitioned into a reducer running on a remote node, then intermediate data shuffling will take place. Let $S(G_i)$ and $T(G_i)$ be a source node and a target node, respectively. We have

$$p_i = \begin{cases} 1, & S(G_i) \neq T(G_i) \\ 0, & \text{Otherwise.} \end{cases} \quad (1)$$

where p_i is set to 0 when the intermediate pair is produced on a local node running the corresponding reduce task; otherwise, p_i is set to 1.

The design goal of FiDoop-DP is to partition transactions in a way to minimize the data transfer cost. Applying (1), we formally express the design goal as:

$$\text{Minimize:} \sum_{i=1}^{m} D_i \times p_i. \quad (2)$$

4 DATA PARTITIONING

FIM is a multi-stage parallel process, where redundant transactions transmission and redundant mining tasks occur in the second MapReduce job. Recall that (see Section 3.1) it is a grand challenge to avoid these downsides by using traditional grouping strategies and default partitioning function. And transferring redundant transactions is a main reason behind high network load and redundant mining cost. To solve this problem, we propose to partition transactions by considering correlations among transactions and items prior to the parallel mining process. That is, transactions with a great similarity are partitioned into one partition in order to prevent the transactions from being repeatedly transmitted to remote nodes. We adopt the Voronoi diagram-based data partitioning technique [19], which is conducive to maintaining data proximity, especially for multi-dimensional data. Therefore, when the second MapReduce job is launched, a new Voronoi diagram-based data partitioning strategy is deployed to minimize unnecessary redundant transaction transmissions.

Voronoi diagram is a way of dividing a space into a number of regions. A set of points referred to as pivots (or seeds) is specified beforehand. For each pivot, there is a corresponding region consisting of all objects closer to it than to the other pivots. The regions are called Voronoi cells. The idea of Voronoi diagram-based partitioning can be formally described as follows. Given a dataset D, Voronoi diagram-based partitioning selects k objects as pivots (donated $p_1, p_2, ..., p_k$). Then, all objects of D are split into k disjoint partitions (donated $C_1, C_2, ..., C_k$), where each object is assigned to the partition with its closest pivot. In this way, the entire data space is split into k cells.

Incorporating the characteristic of FIM, we adopt the similarity as the distance metric between transaction and pivot (or between two transactions) in Voronoi diagram (see Section 4.1 for details). In addition, Voronoi diagram-based partitioning relies on a way of selecting a set of pivots. Thus, in what follows, we investigate distance measure and pivot-selection strategies, followed by partitioning strategies.

4.1 Distance Metric

Recall that to optimize FIM, a good partitioning strategy should cluster similar data objects to the same partition. Similarity is a metric to quantitatively measure the correlation strength between two objects. To capture the characteristics of transactions, we adopt the Jaccard similarity as a distance metric. Jaccard similarity is a statistic commonly

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2560176, IEEE Transactions on Parallel and Distributed Systems

used for comparing the similarity and diversity of sample data objects. A high Jaccard similarity value indicates that two data sets are very close to each other in terms of distance.

In order to quantify the distance among transactions, we model each transaction in a database as a set. Then, the distance among transactions is measured using the Jaccard similarity among these sets. The Jaccard similarity of two sets A and B is defined as

$$\mathrm{J(A,B)} = \frac{|\mathrm{A}\cap\mathrm{B}|}{|\mathrm{A}\cup\mathrm{B}|} \tag{3}$$

Obviously, $J(A, B)$ is a number ranging between 0 and 1; it is 0 when the two sets are disjoint, 1 when they are identical, and strictly between 0 and 1 otherwise. That is, the distance between two sets is close when their Jaccard index is closer to 1; if there is a large distance between the two sets, their Jaccard index is closer to 0.

4.2 K-means Selection of Pivots

Intuitively, selecting pivots directly affects the uniformity coefficient of the remaining objects for voronoi diagram-based partitioning. In particular, we employ the K-means-based selection strategy (see [19]) to choose pivots. And the pivot selecting process is conducted as a data preprocessing phase.

K-means is a popular algorithm for clustering analysis in data mining. K-means clustering aims to partition n objects into k clusters [20][21]. That is, given a set of objects $(x_1, x_2, , x_n)$, where each object is a d-dimensional real vector, k-means clustering partitions the n objects into k ($k \leq n$) sets $C = C_1, C_2, , C_k$, in which each object belongs to a cluster with the nearest mean. The clustering results can be applied to partition the data space into Voronoi cells. To reduce the computational cost of k-means, we perform sampling on the transaction database before running the k-means algorithm. It is worth mentioning that the selection of initial pivots (a.k.a., seeds) plays a critical role in clustering performance. Thus, k-means++ [22]- an extension of k-means, is adopted to conduct pivots selection. After the k data clusters are generated, we choose the center point of each cluster as a pivot for the Voronoi diagram-based data partitioning.

4.3 Partitioning Strategies

Upon the selection of pivots, we calculate the distances from the rest of the objects to these pivots to determine a partition to which each object belongs. We develop the LSH-based strategy to implement a novel grouping and partitioning process, prior to which MinHash is employed as a foundation for LSH.

4.3.1 MinHash

MinHash offers a quick solution to estimate how similar two sets are [23]. MinHash is increasingly becoming a popular solution for large-scale clustering problems. MinHash replaces large sets by much smaller representations called "signatures" composed of "minhash" of the characteristic matrix (i.e., a matrix representation of data sets). Then, MinHash computes an expected similarity of two data sets based on the signatures. Thus, these two phases are detailed below.

First, a characteristic matrix is created from transactions and items in a database. Given a transaction database $D = \{t_1, t_2, ..., t_n\}$, which contains m items. We create an m-by-n characteristic matrix M, where columns represent transactions; rows denote items of the universal item set. Given item r (i.e., a row in the matrix) and transaction c (i.e., a column in the matrix), we set the value in position (r, c) to 1 if item r is a member in transaction c; otherwise, the value of (r, c) is set to 0.

Second, a signature matrix is constructed using the characteristic matrix obtained in the above step. Let h be a hash function mapping members of any set to distinct integers. Given a set $T = \{x_1, ..., x_n\}$, we define $hmin(T)$ to be T's member x, whose hash value (i.e., $h(x)$) is the minimal one among all the hash values of the members in T. Thus, we have

$$hmin(T) = x, \text{where } h(x) = Min_{i=1}^{n}(h(x_i)) \tag{4}$$

We randomly permute, for the first time, the rows of the characteristic matrix. For each column (e.g., c_i representing a transaction), we compute the column's hash value $hmin(c_i)$ using (4). Then, the value in position $(1, i)$ of the signature matrix is set to $hmin(c_i)$. Next, we permute the rows of the characteristic matrix, for a second time, to determine the value in position $(2, i)$ $(1 \leq i \leq n)$. We repeatedly perform the above steps to obtain the value in position (j, i), where j denotes the jth permutation as well as the jth row in the signature matrix; i indicates the ith column in the signature matrix.

Finally, it is necessary to collect multiple (e.g., l) independent MinHash values for each column in M to form an $l \times n$ signatures matrix $M^{'}$. We make use of the signature matrix to calculate the similarity of any pair of two transactions.

Though MinHash is widely applied to estimate the similarity of any pair of two sets, the number of pairs in a large database D is likely to be very big. If we decide to conduct thorough pair-wise comparisons, the computing cost would be unsustainable.

4.3.2 LSH-based Partitioning

Locality sensitive hashing, or LSH, boosts the performance of MinHash by avoiding the comparisons of a large number of element pairs [24][25]. Unlike MinHash repeatedly evaluating an excessive number of pairs, LSH scans all the transactions once to identify all the pairs that are likely to be similar. We adopt LSH to map transactions in the feature space to a number of buckets in a way that similar transactions are likely to be mapped into the same buckets. More formally, the locality sensitive Hash function family is defined as follows.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2560176, IEEE Transactions on Parallel and Distributed Systems

For Hash family H, if any two points p and q satisfy the following conditions, then H is called (R, c, P_1, P_2)-sensitive:

1) If $\| p–q \| \leq R$, then $Pr_H(h(p) = h(q)) \geq P_1$.
2) If $\| p–q \| \leq cR$, then $Pr_H(h(p) = h(q)) \leq P_2$.

A family is interesting when $P_1 > P_2$.

The above condition 1) ensures two similar points are mapped into the same buckets with a high probability; condition 2) guarantees two dissimilar points are less likely to be mapped into the same buckets.

LSH has to make use of the MinHash signature matrix obtained in 4.3.1 (i.e., M'). Given the $l \times n$ signature matrix M', we design an effective way of choosing the hash family by dividing the signature matrix into b bands consisting of r rows, where $b \times r = l$. For each band, there is a hash function that takes the r integers (the portion of one column within that band) as a vector, which is placed into a hash bucket.

It relies on the use of a family of locality preserving hash functions, creating several hash tables that similar items with high probability are more likely to be hashed into the same bucket than dissimilar items [26]. From the way of establishing Hash Table, we obtain that the time complexity of lookup is O(1).

5 IMPLEMENTATION DETAILS

In this section, we present the implementation details of LSH-based FiDoop-DP running on Hadoop clusters. Please refer to Fig.2 for FiDoop-DP's processing flow, which consists of four steps (i.e., one sequential-computing step and three parallel MapReduce jobs) (see Section 2.3). Specifically, before launching the FiDoop-DP process, a preprocessing phase is performed in a master node to select a set of (k) pivots which serve as an input of the second MapReduce job that is responsible for the Voronoi diagram-based partitioning (see Section 4.2).

In the first MapReduce job, each mapper sequentially reads each transaction from its local input split on a data node to generate local 1-itemsets. Next, global 1-itemsets are produced by a specific reducer, which merges local 1-itemsets sharing the same key (i.e., item name). The output of these reducers include the global frequent 1-itemsets along with their counts. The second step sorts these global frequent 1-itemsets in a decreasing order of frequency; the sorted frequent 1-itemsets are saved in a cache named $FList$, which becomes an input of the second MapReduce job in FiDoop-DP.

The second MapReduce job applies a second-round scanning on the database to repartition database to form a complete dataset for item groups in the map phase. Each reducer conducts local FP-Growth based on the partitions to generate all frequent patterns.

The last MapReduce job aggregates the second MapReduce job's output (i.e., all the frequent patterns) to generate the final frequent patterns for each item. For example, the output of the second MapReduce job includes three frequent patterns, namely, 'abc', 'adc', and 'bdc'. Using these three frequent patterns as an input, the third MapReduce job creates the final results for each item as 'a: abc,adc', 'b: abc,bdc', 'c: abc,adc,bdc', and 'd: adc,bdc'.

We pay attention to the second MapReduce job and the reason is three-fold. First, at the heart of FiDoop-DP is the construction of all frequent patterns, which is implemented in the second MapReduce job. Second, this MapReduce job is more complicated and comprehensive than the first and the third ones. Third, this job plays a vital role in achieving high performance of FiDoop-DP. To optimize the performance of Pfp, we make an improvement in the second MapReduce job by incorporating the Voronoi diagram-based partitioning idea. In what follows, we elaborate the algorithm for the second MapReduce job.

Algorithm 1 LSH-Fpgrowth

Input: $FList$, k pivots, DB_i;
Output: transactions corresponding to each Gid;
function MAP(key offset, values DB_i)
  load $FList$, k pivots;
  $Glists \leftarrow GenerateGlists(FList, kpivots)$;/* based on the correlation of each item in FList and k pivots */
  for all (T in DB_i) **do**
    $items[] \leftarrow Split(eachT)$;
    for all (item in items[]) **do**
      if item is in FList **then**
        $a[] \leftarrow item$
      end if
    end for
    Add Generate-signature-matrix(a[]) into Arrarylist $sigMatrix$;
  end for
  for all (c_i in $sigMatrix$) **do**
    divide c_i into b bands with r rows;
    $Hashbucket \leftarrow HashMap(each\ band\ of\ c_i())$;
  end for
  if at least one band of c_i and pivot p_j is hashed into the same bucket **then**
    $Gid \leftarrow j$;
    Output(Gid, new TransactionTree(a[i]));
  end if
  for all each $GList_t (t \neq i)$ **do**
    if c_i contains an item in $GList_t$ **then**
      $Gid \leftarrow t$
      Output(Gid, new TransactionTree(a[i])); /* guarantee the data completeness for each GList */
    end if
  end for
end function

Input: transactions corresponding to each Gid;
Output: frequent k-itemsets;
function REDUCE(key Gid, values DB_{Gid})
  Load GLists;
  $nowGroup \leftarrow GList_{Gid}$
  localFptree.clear;
  for all (T_i in DB_{Gid}) **do**
    insert-build-fp-tree(localFptree, T_i);
  end for
  for all (a_i in nowGroup) **do**
    Define a max heap HP with size K;
    Call TopKFPGrowth(localFptree,a_i,HP);
    for all (v_i in HP) **do**
      Output(v_i, support(v_i));
    end for
  end for
end function

Given a set of k pivots $(p_1, p_2, ..., p_k)$ selected in the preprocessing step, we perform item grouping and data partitioning using statistical data collected for each partition. Algorithm 1 is an LSH-based approach that integrates

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2560176, IEEE Transactions on Parallel and Distributed Systems

the item grouping (see Step 3) and partitioning processes (see Steps 4-20).

In Algorithm 1, each mapper takes transactions as an input in the format of $Pair\langle LongWritableoffset, Textrecord\rangle$ (see Step 1). The mappers concurrently load $FList$ to filter infrequent items of each transaction (see Step 2). Meanwhile, $FList$ is divided into Q groups (i.e., *GLists*) by determining similarity among items and the given pivots $(P_1, P_2, ..., P_k)$; each $GList$ consists of Gid and the collection of items in the group (see Step 3). Then, each "record", including the pivots $(P_1, P_2, ..., P_k)$, T_i is transformed into a set, followed by applying the minhash function to generate a column c_i of signatures matrix (see Steps 4-12 and algorithm 2). LSH is carried out using the above signature matrix M' (l×n) (see Steps 13-16). M' is divided into b bands, each of which contains r rows (where b×$r = l$). Then, these bands are hashed to a number of hash buckets; each hash bucket contains similar transactions (see Step 15).

Below we show the rationale behind applying LSH to determine similarity among transactions. Given two transactions (e.g., T_1 and T_2), if there exists at least a pair of bands (e.g., $b_1 \in T_1$ and $b_2 \in T_2$) such that bands b_1 and b_2 are hashed into the same bucket, then transactions T_1 and T_2 are considered similar (see Step 17). Assume the similarity between two columns (denoted as $c1, c2$) of a signature matrix is p, then the probability that $c1$ and $c2$ are exactly the same in a band is p^r; the probability that $c1$ and $c2$ are completely different with respect to all the b bands is $1–s^r$. We show that if selecting appropriate values of b and r, transactions with a great similarity are mapped into one bucket with a very high probability.

If a band of T_i shares the same bucket with a band of P_j, we assign T_i to the partition labelled as P_j. We donate such an assignment in form of a pair $Pair\langle P_j, T_i\rangle$) (see Steps 18-19). At the end of the map tasks, $GLists$ are checked to guarantee the data completeness (Steps 21-24).

Finally, the mappers emit $Pair\langle P_i, T_i\rangle$ to be shuffled and combined for the second job's reducers, and reducers conduct local FP-Growth to generate the final frequent patterns of each item (see Steps 28-42).

During the process of generating the signature matrix, it is infeasible to permute a large characteristic matrix due to high time complexity. This problem is addressed by employing the Minwise Independent permutation [27] to speed up the process (see algorithm 2). Let *h(x)* be a permutation function on a set X, for an element $x \subseteq X$, the value permuted is $h(x) = min(h(x_1), h(x_2), ..., h(x_n))$. When we obtain the signature matrix, the original high-dimensional data are mapped to a low-dimensional space. And the time complexity of subsequent operations is greatly reduced thanks to the above dimensions reduction.

Algorithm 2 Generate-signature-matrix

```
Input: a[];
Output: signature matrix of a[];
 1: function GENERATE-SIGNATURE-MATRIX(a[])
 2:     for (i=0;i < numHashFunctions;i++) do
 3:         minHashValues[i] = Integer.MAX_VALUE;
 4:     end for
 5:     for (i=0;i < numHashFunctions;i++) do
 6:         for all ele: a[] do
 7:             value ← Integer(ele);
 8:             bytesToHash[0]=(byte)(value >> 24);
 9:             bytesToHash[1]=(byte)(value >> 16);
10:             bytesToHash[2]=(byte)(value >> 8);
11:             bytesToHash[3]=(byte)value);
12:             hashIndex ← hashFunction[i].hash(bytesToHash);
13:             if (minHashValues[i]) > hashIndex then
14:                 minHashValues[i]=hashIndex;
15:             end if
16:         end for
17:     end for
18: end function
```

6 EXPERIMENTAL EVALUATION

We implement and evaluate the performance of FiDoop-DP on our in-house Hadoop cluster equipped with 24 data nodes. Each node has an Intel E5-1620 v2 series 3.7gHZ 4 core processor, 16G main memory, and runs on the Centos 6.4 operating system, on which Java JDK 1.8.0_20 and Hadoop 1.1.2 are installed. The hard disk of NameNode is configured to 500GB; and the capacity of disks in each DataNode is 2TB. All the data nodes of the cluster have Gigabit Ethernet NICs connected to Gigabit ports on the switch; the nodes can communicate with one another using the SSH protocol. We use the default Hadoop parameter configurations to set up the replication factor (i.e., three) and the numbers of Map and Reduce tasks. Our experimental results show that over 90% of the processing time is spent running the second MapReduce job; therefore, we focus on performance evaluation of this job in our experiments.

To evaluate the performance of the proposed FiDoop-DP, We generate synthetic datasets using the IBM Quest Market-Basket Synthetic Data Generator [28], which can be flexibly configured to create a wide range of data sets to meet the needs of various test requirements. The parameters' characteristics of our dataset are summarized in Table I.

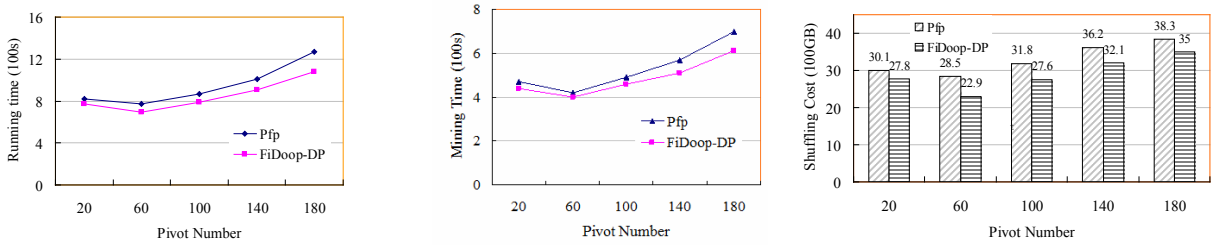
TABLE I. Dataset

Parameters	Avg.length	#Items	Avg.Size/Transaction
T10I4D	10	4000	17.5B
T40I10D	40	10000	31.5B
T60I10D	60	10000	43.6B
T85I10D	85	10000	63.7B

6.1 The Number of Pivots

We compare the performance of FiDoop-DP and Pfp [11] when the number k of pivots varies from 20 to 180. Please note that k in FiDoop-DP corresponds to the number of groups in Pfp. Fig. 3 reveals the running time, shuffling cost, and mining cost of FiDoop-DP and Pfp processing the 4G 61-block T40I10D dataset on the 8-node cluster. Fig. 3 shows that FiDoop-DP improves the overall performance of Pfp. Such performance improvements are contributed by good data locality achieved by Fidoop-DP's analysis of correlation among the data. FiDoop-DP optimizes data

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2560176, IEEE Transactions on Parallel and Distributed Systems



(a) Running Time (b) Mining Cost (c) Shuffling Cost

Fig. 3. Impacts of the number of pivots on FiDoop-DP and Pfp.

locality to reduce network and computing loads by eliminating of redundant transactions on multiple nodes. As a result, FiDoop-DP is capable of cutting mining cost (see Fig. 3(b)) and data shuffling cost (see Fig. 3(c)).

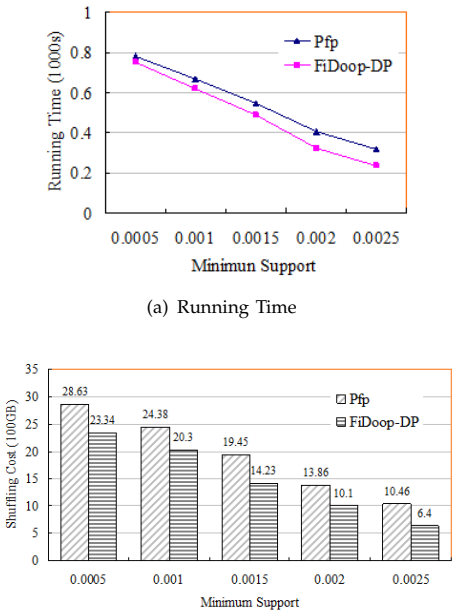
Fig. 3(a) illustrates that the performance improvement of FiDoop-DP over Pfp becomes pronounced when the number k of pivots is large (e.g., 180). A large k in Pfp gives rise to a large number of groups, which in turn leads to an excessive number of redundant transactions processed and transfers among data nodes. As such, the large k offers a great opportunity for FiDoop-DP to alleviate Pfp's heavy CPU and network loads induced by the redundant transactions.

Interestingly, we observe from Fig. 3(a) that the overall running times of the two algorithms are minimized when number k is set to 60. Such minimized running times are attributed to (1) the FP-Growth mining cost plotted in Fig. 3(b) and (2) the shuffling cost shown in Fig. 3(c). Figs. 3(b) and 3(c) illustrate that the mining cost and shuffling cost are minimized when parameter k becomes 60 in a range from 20 to 180.

The running times, mining cost, and shuffling cost exhibit a U-shape in Fig. 3 because of the following reasons. To conduct the local FP-Growth algorithm, we need to group frequent 1-itemsets followed by partitioning transactions based on items contained in each item group. When the number of pivots increases, the entire database is split into a finer granularity and the number of partitions increase correspondingly. Such a fine granularity leads to a reduction in distance computation among transactions. On the other hand, when the pivot number k continues growing, the number of transactions mapped into one hash bucket significantly increases, thereby leading to a large candidate-object set and high shuffling cost (see Figs. 3(b) and 3(c)). Consequently, the overall execution time is optimized when k is 60 for both algorithms (see Fig. 3(a)).

6.2 Minimum Support

Recall that minimum support plays an important role in mining frequent itemsets. We increase minimum support thresholds from 0.0005% to 0.0025% with an increment of 0.0005% to evaluate the impact of minimum support on FiDoop-DP. The other parameters are the same as those for the previous experiments.



(a) Running Time

(b) Shuffling Cost

Fig. 4. Impact of minimum support on FiDoop-DP and Pfp.

Fig. 4(a) shows that the execution times of FiDoop-DP and Pfp decrease when the minimum support is increasing. Intuitively, a small minimum support leads to an increasing number of frequent 1-itemsets and transactions, which have to be scanned and transmitted. Table II illustrates the size of frequent 1-itemsets stored in $FList$ and the number of final output records of the two parallel solutions under various minimum-support values.

TABLE II. The size of $FList$ and the number of final output records under various minimum-support values.

minsupport	0.0005%	0.001%	0.0015%	0.002%	0.0025%
FList	14.69k	11.6k	9.71k	6.89k	5.51k
OutRecords	745	588	465	348	278

Fig. 4(a) reveals that regardless of the minimum-support value, FiDoop-DP is superior to Pfp in terms of running time. Two reasons make this performance trend expected. First, FiDoop-DP optimizes the partitioning process by placing transactions with a high similarity into one group rather than randomly and evenly grouping the transaction. Fig. 4(b) confirms that FiDoop-DP's shuffling cost is significantly lower than that of Pfp thanks to optimal data partitions offered by FiDoop-DP. Second, this grouping strategy

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2560176, IEEE Transactions on Parallel and Distributed Systems

in FiDoop-DP minimizes the number of transactions for each $GList$ under the premise of data completeness, which leads to reducing mining load for each Reducer. The grouping strategy of FiDoop-DP introduces computing overhead including signature-matrix calculation and hashing each band into a bucket. Nevertheless, such small overhead is offset by the performance gains in the shuffling and reduce phases.

Fig. 4(a) also shows that the performance improvement of FiDoop-DP over Pfp is widened when the minimum support increases. This performance gap between FiDoop-DP and Pfp is reasonable, because pushing minimum support up in FiDoop-DP filters out an increased number of frequent 1-itemsets, which in turn shortens the transaction partitioning cost. Small transactions simplify the correlation analysis among the transactions; thus, small transactions are less likely to have a large number of duplications in their partitions. As a result, the number of duplicated transactions to be transmitted among the partitions is significantly reduced, which allows FiDoop-DP to deliver better performance than Pfp.

6.3 Data Characteristic

In this group of experiments, we respectively evaluate the impact of dimensionality and data correlation on the performance of FiDoop-DP and Pfp by changing the parameters in the process of generating the datasets using the IBM Quest Market-Basket Synthetic Data Generator.

6.3.1 Dimensionality

The average transaction length directly determines the dimensions of a test data. We configure the average transaction length to 10, 40, 60, and 85 to generate T10I4D (130 blocks), T40I10D (128 blocks), T60I10D (135 blocks), T85I10D (133 blocks) datasets, respectively. In this experiment, we measure the impacts of dimensions on the performance of FiDoop-DP and Pfp on the 8-node Hadoop cluster.

The experimental results plotted in Fig. 5(a) clearly indicate that an increasing number of dimensions significantly raises the running times of FiDoop-DP and Pfp. This is because increasing the number of dimensions increases the number of groups; thus, the amount of data transmission sharply goes up as seen in Fig. 5(b).

The performance improvements of FiDoop-DP over Pfp is diminishing when the dimensionality increases from 10 to 85. For example, FiDoop-DP offers an improvement of 29.4% when the dimensionality is set to 10; the improvement drops to 5.2% when the number of dimensions becomes 85.

In what follows, we argue that FiDoop-DP is inherently losing the power of reducing the number of redundant transactions in high-dimensional data. When a dataset has a low dimensionality, FiDoop-DP tends to build partitions, each of which has distinct characteristics compared with the other partitions. Such distinct features among the partitions allow FiDoop-DP to efficiently reduce the number of redundant transactions. In contrast, a dataset with high dimensionality has a long average transaction length; therefore, data partitions produced by FiDoop-DP have no distinct discrepancy. Redundant transactions are likely to be formed for partitions that lack distinct characteristics. Consequently, the benefit offered by FiDoop-DP for high-dimensional datasets becomes insignificant.

6.3.2 Data correlation

We set the correlation among transactions (i.e., -corr) to 0.15, 0.25, 0.35, 0.45, 0.55, 0.65 and 0.75 to measure the impacts of data correlation on the performance of the two algorithms on the 8-node Hadoop cluster. The Number of Pivots is set to 60 (see also Section 6.1).

The experimental results plotted in Fig. 5(c) clearly indicate that FiDoop-DP is more sensitive to data correlation than Pfp. This performance trend motivates us to investigate the correlation-related data partition strategy. Pfp conducts default data partition based on equal-size item group without taking into account the characteristics of the datasets. However, FiDoop-DP judiciously groups items with high correlation into one group and clustering similar transactions together. In this way, the number of redundant transactions kept on multiple nodes is substantially reduced. Consequently, FiDoop-DP is conducive to cutting back both data transmission traffic and computing load.

As can be seen from Fig. 5(c), there is an optimum balance point for data correlation degree to tune FiDoop-DP performance (e.g., 0.35 in Fig. 5(c)). If data correlation is too small, Fidoop-DP will degenerate into random partition schema. On the contrary, it is difficult to divide items into relatively independent groups when data correlation is high, meaning that an excessive number of duplicated transactions have to be transferred to multiple nodes. Thus, a high data correlation leads to redundant transactions formed for partitions, thereby increasing network and computing loads.

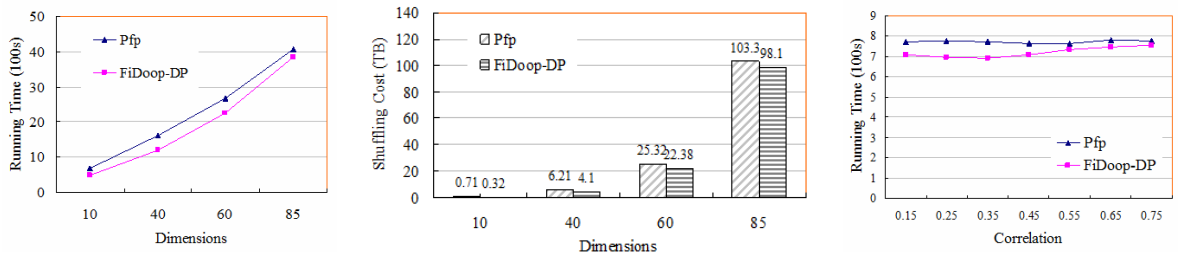
6.4 Speedup

Now we are positioned to evaluate the speedup performance of FiDoop-DP and Pfp by increasing the number of data nodes in our Hadoop cluster from 4 to 24. The T40I10D (128 blocks) dataset is applied to drive the speedup analysis of the these algorithms. Fig. 6 reveals the speedups of FiDoop-DP and Pfp as a function of the number of data nodes.

The experimental results illustrated in Fig. 6(a) show that the speedups of FiDoop-DP and Pfp linearly scale up with the increasing number of data nodes. 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 among mappers and reducers.

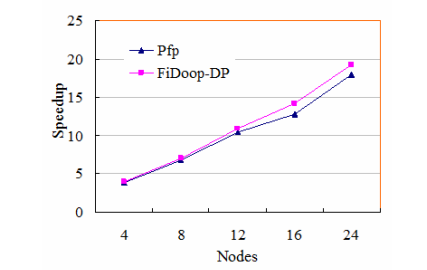
Fig. 6(a) shows that FiDoop-DP is better than Pfp in terms of the speedup efficiency. For instance, the FiDoop-DP improves the speedup efficiency of Pfp by up to 11.2%

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2560176, IEEE Transactions on Parallel and Distributed Systems

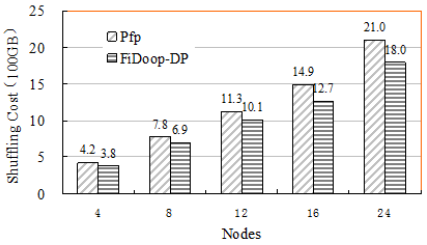


(a) Running Time (b) Shuffling Cost (c) Running Time

Fig. 5. Impacts of data characteristics on FiDoop-DP and Pfp.



(a) Speedup



(b) Shuffling Cost

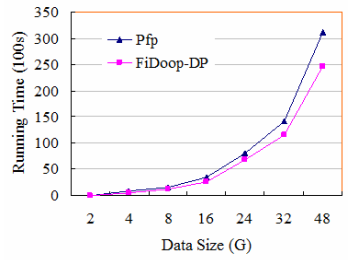
Fig. 6. The speedup performance and shuffling cost of FiDoop-DP and Pfp.

with an average of 6.1%. This trend suggests FiDoop-DP improves the speedup efficiency of Pfp in large-scale

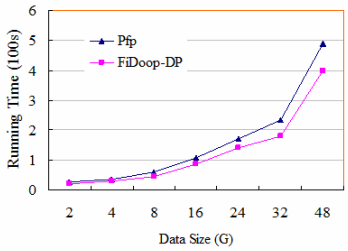
The speedup efficiencies drop when the Hadoop cluster scales up. For example, the speedup efficiencies of FiDoop-DP and Pfp on the 4-node cluster are 0.970 and 0.995, respectively. These two speedup efficiencies become 0.746 and 0.800 on the 24-node cluster. Such a speedup-efficiency trend is driven by the cost of shuffling intermediate results, which sharply goes up when the number of data nodes scales up. 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. 6(b) confirm that the shuffling cost is linearly increasing when computing nodes are scaled from 4 to 24. Furthermore, the shuffling cost of Pfp is larger than that of FiDoop-DP.

6.5 Scalability

In this group of experiments, we evaluate the scalability of FiDoop-DP and Pfp when the size of input dataset dramatically grows. Fig. 7 shows the running times of the algorithms when we scale up the size of the T40I10D data series. Figs. 7(a) and 7(b) demonstrate the performance of FiDoop-DP processing various datasets on 8-node and 24-node clusters,



(a) On 8 nodes



(b) On 24 nodes

Fig. 7. The scalability of FiDoop-DP and Pfp when the size of input dataset increases.

Fig. 7 clearly reveals that the overall execution times of FiDoop-DP and Pfp go 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 size leads to long scanning time. Interestingly, FiDoop-DP exhibits a better scalability than Pfp.

Recall that (see also from Algorithm 1) the second MapReduce job compresses an initial transaction database into a signature matrix, which is dealt by the subsequent process. The compress ratio is high when the input data size is large, thereby shortening the subsequent processing time. Furthermore, Fidoop-DP lowers the network traffic induced by the random grouping strategy in Pfp. In summary, the scalability of FiDoop-DP is higher than that of Pfp when it comes to parallel mining of an enormous amount of data.

7 RELATED WORK

7.1 Data Partitioning in MapReduce

Partitioning in databases has been widely studied, for both single system servers (e.g. [29]) and distributed storage

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2560176, IEEE Transactions on Parallel and Distributed Systems

systems (e.g., BigTable [30], PNUTS[31]). The existing approaches typically produce possible ranges or hash partitions, which are then evaluated using heuristics and cost models. These schemes offer limited support for OLTP workloads or query analysis in the context of the popular MapReduce programming model. In this study, we focus on the data partitioning issue in MapReduce.

High scalability is one of the most important design goals for MapReduce applications. Unfortunately, the partitioning techniques in existing MapReduce platforms (e.g., Hadoop) are in their infancy, leading to serious performance problems.

Recently, a handful of data partitioning schemes have been proposed in the MapReduce platforms. Xie *et al.* developed a data placement management mechanism for heterogeneous Hadoop clusters. Their mechanism partitions data fragments to nodes in accordance to the nodes' processing speed measured by computing ratios [32]. In addition, Xie *et al.* designed a data redistribution algorithm in HDFS to address the data-skew issue imposed by dynamic data insertions and deletions. CoHadoop [33] is a Hadoop's lightweight extension, which is designed to identify related data files followed by a modified data placement policy to co-locate copies of those related files in the same server. CoHadoop considers the relevance among files; that is, CoHadoop is an optimization of HaDoop for multiple files. A key assumption of the MapReduce programming model is that mappers are completely independent of one another. Vernica *et al.* broke such an assumption by introducing an asynchronous communication channel among mappers [34]. This channel enables the mappers to see global states managed in metadata. Such situation-aware mappers (SAMs) can enable MapReduce to flexibly partition the inputs. Apart from this, adaptive sampling and partitioning were proposed to produce balanced partitions for the reducers by sampling mapper outputs and making use of obtained statistics.

Graph and hypergraph partitioning have been used to guide data partitioning in parallel computing. Graph-based partitioning schemes capture data relationships. For example, Ke *et al.* applied a graphic-execution-plan graph (EPG) to perform cost estimation and optimization by analyzing various properties of both data and computation [35]. Their estimation module coupled with the cost model estimate the runtime cost of each vertex in an EPG, which represents the overall runtime cost; a data partitioning plan is determined by a cost optimization module. Liroz-Gistau *et al.* proposed the MR-Part technique, which partitions all input tuples producing the same intermediate key co-located in the same chunk. Such a partitioning approach minimizes data transmission among mappers and reducers in the shuffle phase [36]. The approach captures the relationships between input tuples and intermediate keys by monitoring the execution of representative workload. Then, based on these relationships, their approach applies a min-cut k-way graph partitioning algorithm, thereby partitioning and assigning the tuples to appropriate fragments by modeling the workload with a hyper graph. In doing so, subsequent MapReduce jobs take full advantage of data locality in the reduce phase. Their partitioning strategy suffers from adverse initialization overhead.

7.2 Application-Aware Data Partitioning

Various efficient data partitioning strategies have been proposed to improve the performance of parallel computing systems. For example, Kirsten *et al.* developed two general partitioning strategies for generating entity match tasks to avoid memory bottlenecks and load imbalances [37]. Taking into account the characteristics of input data, Aridhi *et al.* proposed a novel density-based data partitioning technique for approximate large-scale frequent subgraph mining to balance computational load among a collection of machines. Kotoulas *et al.* built a data distribution mechanism based on clustering in elastic regions [38].

Traditional term-based partitioning has limited scalability due to the existence of very skewed frequency distributions among terms. Load-balanced distributed clustering across networks and local clustering are introduced to improve the chance that triples with a same key are collocated. These self-organizing approaches need no data analysis or upfront parameter adjustments in a priori. Lu *et al.* studied k nearest neighbor join using MapReduce, in which a data partitioning approach was designed to reduce both shuffling and computational costs [19]. In Lu's study, objects are divided into partitions using a Voronoi diagram with carefully selected pivots. Then, data partitions (i.e., Voronoi cells) are clustered into groups only if distances between them are restricted by a specific bound. In this way, their approach can answer the k-nearest-neighbour join queries by simply checking object pairs within each group.

FIM for data-intensive applications over computing clusters has received a growing attention; efficient data partitioning strategies have been proposed to improve the performance of parallel FIM algorithms. A MapReduce-based Apriori algorithm is designed to incorporate a new dynamic partitioning and distributing data method to improve mining performance [39]. This method divides input data into relatively small splits to provide flexibility for improved load-balance performance. Moreover, the master node doesn't distribute all the data once; rather, the rest data are distributed based on dynamically changing workload and computing capability weight of each node. Similarly, Jumbo [40] adopted a dynamic partition assignment technology, enabling each task to process more than one partition. Thus, these partitions can be dynamically reassigned to different tasks to improve the load balancing performance of Pfp [11]. Uthayopas *et al.* investigated I/O and execution scheduling strategies to balance data processing load, thereby enhancing the utilization of a multicore cluster system supporting association-rule mining. In order to pick a winning strategy in terms of data-blocks assignment, Uthayopas *et al.* incorporated three basic placement policies, namely, the round robin, range, and random placement. Their approach ignores data characteristics during the course of mining association rules.

8 FURTHER DISCUSSIONS

In this study, we investigated the data partitioning issues in parallel FIM. We focused on MapReduce-based parallel FP-tree algorithms; in particular, we studied how to partition and distribute a large dataset across data nodes of a Hadoop cluster to reduce network and computing loads.

We argue that the general idea of FiDoop-DP proposed in this study can be extended to other FIM algorithms like Apriori running on Hadoop clusters. Apriori-based parallel FIM algorithms can be classified into two camps, namely, *count distribution* and *data distribution* [41]. For the *count distribution* camp, each node in a cluster calculates local support counts of all candidate itemsets. Then, the global support counts of the candidates are computed by exchanging the local support counts. For the *data distribution* camp, each node only keeps the support counts of a subset of all candidates. Each node is responsible for delivering its local database partition to all the other processors to compute support counts. In general, the *data distribution* schemes have higher communication overhead than the *count distribution* ones; whereas the *data distribution* schemes have lower synchronization overhead than its competitor.

Regardless of the *count distribution* or *data distribution* approaches, the communication and synchronization cost induce adverse impacts on the performance of parallel mining algorithms. The basic idea of Fidoop-DP - grouping highly relevant transactions into a partition - allows the parallel algorithms to exploit correlations among transactions in database to cut communication and synchronization overhead among Hadoop nodes.

9 CONCLUSIONS AND FUTURE WORK

To mitigate high communication and reduce computing cost in MapReduce-based FIM algorithms, we developed FiDoop-DP, which exploits correlation among transactions to partition a large dataset across data nodes in a Hadoop cluster. FiDoop-DP is able to (1) partition transactions with high similarity together and (2) group highly correlated frequent items into a list. One of the salient features of FiDoop-DP lies in its capability of lowering network traffic and computing load through reducing the number of redundant transactions, which are transmitted among Hadoop nodes. FiDoop-DP applies the Voronoi diagram-based data partitioning technique to accomplish data partition, in which LSH is incorporated to offer an analysis of correlation among transactions. At the heart of FiDoop-DP is the second MapReduce job, which (1) partitions a large database to form a complete dataset for item groups and (2) conducts FP-Growth processing in parallel on local partitions to generate all frequent patterns. Our experimental results reveal that FiDoop-DP significantly improves the FIM performance of the existing Pfp solution by up to 31% with an average of 18%.

We introduced in this study a similarity metric to facilitate data-aware partitioning. As a future research direction, we will apply this metric to investigate advanced load-balancing strategies on a heterogeneous Hadoop cluster. In one of our earlier studies (see [32] for details), we 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 [32] can balance the amount of data stored in heterogeneous nodes to achieve improved data-processing performance. Such a scheme implemented at the level of Hadoop distributed file system (HDFS) is unaware of correlations among application data. To further improve load balancing mechanisms implemented in HDFS, we plan to integrate FiDoop-DP with a data-placement mechanism in HDFS on heterogeneous clusters. In addition to performance issues, energy efficiency of parallel FIM systems will be an intriguing research direction.

ACKNOWLEDGMENT

The work in this paper was in part supported by the National Natural Science Foundation of P.R. China (No.61272263, No.61572343). Xiao Qin's work was supported by the U.S. National Science Foundation under Grants CCF-0845257 (CAREER). The authors would also like to thank Mojen Lau for proof-reading.

REFERENCES

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

[2] I. Pramudiono and M. Kitsuregawa, "Fp-tax: Tree structure based generalized association rule mining," in *Proceedings of the 9th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*. ACM, 2004, pp. 60–63.

[3] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[4] S. Sakr, A. Liu, and A. G. Fayoumi, "The family of mapreduce and large-scale data processing systems," *ACM Computing Surveys (CSUR)*, vol. 46, no. 1, p. 11, 2013.

[5] M.-Y. Lin, P.-Y. Lee, and S.-C. Hsueh, "Apriori-based frequent itemset mining algorithms on mapreduce," in *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, ser. ICUIMC '12. New York, NY, USA: ACM, 2012, pp. 76:1–76:8.

[6] X. Lin, "Mr-apriori: Association rules algorithm based on mapreduce," in *Software Engineering and Service Science (ICSESS), 2014 5th IEEE International Conference on*. IEEE, 2014, pp. 141–144.

[7] L. Zhou, Z. Zhong, J. Chang, J. Li, J. Huang, and S. Feng, "Balanced parallel fp-growth with mapreduce," in *Information Computing and Telecommunications (YC-ICT), 2010 IEEE Youth Conference on*. IEEE, 2010, pp. 243–246.

[8] S. Hong, Z. Huaxuan, C. Shiping, and H. Chunyan, "The study of improved fp-growth algorithm in mapreduce," in *1st International Workshop on Cloud Computing and Information Security*. Atlantis Press, 2013.

[9] M. Riondato, J. A. DeBrabant, R. Fonseca, and E. Upfal, "Parma: a parallel randomized algorithm for approximate association rules mining in mapreduce," in *Proceedings of the 21st ACM international conference on Information and knowledge management*. ACM, 2012, pp. 85–94.

[10] C. Lam, *Hadoop in action*. Manning Publications Co., 2010.

[11] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang, "Pfp: parallel fp-growth for query recommendation," in *Proceedings of the 2008 ACM conference on Recommender systems*. ACM, 2008, pp. 107–114.

[12] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 48–57, 2010.

[13] P. Uthayopas and N. Benjamas, "Impact of i/o and execution scheduling strategies on large scale parallel data mining," *Journal of Next Generation Information Technology (JNIT)*, vol. 5, no. 1, p. 78, 2014.

[14] I. Pramudiono and M. Kitsuregawa, "Parallel fp-growth on pc cluster," in *Advances in Knowledge Discovery and Data Mining*. Springer, 2003, pp. 467–473.

[15] Y. Xun, J. Zhang, and X. Qin, "Fidoop: Parallel mining of frequent itemsets using mapreduce," *IEEE Transactions on Systems, Man, and Cybernetics: Systems, doi: 10.1109/TSMC.2015.2437327*, 2015.

[16] S. Owen, R. Anil, T. Dunning, and E. Friedman, *Mahout in action*. Manning, 2011.

[17] D. Borthakur, "Hdfs architecture guide," *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*, 2008.

[18] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, vol. 10, 2010, p. 10.

[19] W. Lu, Y. Shen, S. Chen, and B. C. Ooi, "Efficient processing of k nearest neighbor joins using mapreduce," *Proceedings of the VLDB Endowment*, vol. 5, no. 10, pp. 1016–1027, 2012.

[20] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An efficient k-means clustering algorithm: Analysis and implementation," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 24, no. 7, pp. 881–892, 2002.

[21] A. K. Jain, "Data clustering: 50 years beyond k-means," *Pattern Recognition Letters*, vol. 31, no. 8, pp. 651–666, 2010.

[22] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035.

[23] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of massive datasets*. Cambridge University Press, 2014.

[24] A. Stupar, S. Michel, and R. Schenkel, "Rankreduce–processing k-nearest neighbor queries on top of mapreduce," in *Proceedings of the 8th Workshop on Large-Scale Distributed Systems for Information Retrieval*. Citeseer, 2010, pp. 13–18.

[25] B. Bahmani, A. Goel, and R. Shinde, "Efficient distributed locality sensitive hashing," in *Proceedings of the 21st ACM international conference on Information and knowledge management*. ACM, 2012, pp. 2174–2178.

[26] R. Panigrahy, "Entropy based nearest neighbor search in high dimensions," in *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*. ACM, 2006, pp. 1186–1195.

[27] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, "Min-wise independent permutations," *Journal of Computer and System Sciences*, vol. 60, no. 3, pp. 630–659, 2000.

[28] L. Cristofor, "Artool," 2006.

[29] S. Agrawal, V. Narasayya, and B. Yang, "Integrating vertical and horizontal partitioning into automated physical database design," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 2004, pp. 359–370.

[30] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed structured data storage system," in *7th OSDI*, 2006, pp. 305–314.

[31] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1277–1288, 2008.

[32] J. Xie and X. Qin, "The 19th heterogeneity in computing workshop (hcw 2010)," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, april 2010, pp. 1 –5.

[33] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson, "Cohadoop: flexible data placement and its exploitation in hadoop," *Proceedings of the VLDB Endowment*, vol. 4, no. 9, pp. 575–585, 2011.

[34] R. Vernica, A. Balmin, K. S. Beyer, and V. Ercegovac, "Adaptive mapreduce using situation-aware mappers," in *Proceedings of the 15th International Conference on Extending Database Technology*. ACM, 2012, pp. 420–431.

[35] Q. Ke, V. Prabhakaran, Y. Xie, Y. Yu, J. Wu, and J. Yang, "Optimizing data partitioning for data-parallel computing," Dec. 13 2011, uS Patent App. 13/325,049.

[36] M. Liroz-Gistau, R. Akbarinia, D. Agrawal, E. Pacitti, and P. Valduriez, "Data partitioning for minimizing transferred data in mapreduce," in *Data Management in Cloud, Grid and P2P Systems*. Springer, 2013, pp. 1–12.

[37] T. Kirsten, L. Kolb, M. Hartung, A. Groß, H. Köpcke, and E. Rahm, "Data partitioning for parallel entity matching," *Proceedings of the VLDB Endowment*, vol. 3, no. 2, 2010.

[38] S. Kotoulas, E. Oren, and F. Van Harmelen, "Mind the data skew: distributed inferencing by speeddating in elastic regions," in *Proceedings of the 19th international conference on World wide web*. ACM, 2010, pp. 531–540.

[39] L. Li and M. Zhang, "The strategy of mining association rule based on cloud computing," in *Business Computing and Global Informatization (BCGIN), 2011 International Conference on*. IEEE, 2011, pp. 475–478.

[40] S. Groot, K. Goda, and M. Kitsuregawa, "Towards improved load balancing for data intensive distributed computing," in *Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM, 2011, pp. 139–146.

[41] M. Z. Ashrafi, D. Taniar, and K. Smith, "Odam: An optimized distributed association rule mining algorithm," *IEEE distributed systems online*, no. 3, p. 1, 2004.

Yaling Xun is currently a doctoral student at Taiyuan University of Science and Technology (TYUST). She is currently a lecturer in the School of Computer Science and Technology at TYUST. Her research interests include data mining and parallel computing.

Jifu Zhang received the BS and MS in Computer Science and Technology from Hefei University of Tchnology, China, and the Ph.D. degree in Pattern Recognition and Intelligence Systems from Beijing Institute of Technology in 1983, 1989, and 2005. He is currently a Professor in the School of Computer Science and Technology at TYUST. His research interests include data mining, parallel and distributed computing and artificial intelligence.

Xiao Qin received the Ph.D. degree in Computer Science from the University of Nebraska-Lincoln in 2004. He is a professor in the Department of Computer Science and Software Engineering, Auburn University. His research interests include parallel and distributed systems, storage systems, fault tolerance, real-time systems, and performance evaluation. He received the U.S. NSF Computing Processes and Artifacts Award and the NSF Computer System Research Award in 2007 and the NSF CAREER Award in 2009.

Xujun Zhao received the MS in Computer Science and Technology in 2005 from Taiyuan University of Technology(TYUT), China. He is currently a Ph.D. student at Taiyuan University of Science and Technology(TYUST). His research interests include data mining and parallel computing.