Parallel Hierarchical Subspace Clustering of Categorical Data

Ning Pang, Jifu Zhang, *Member, IEEE*, Chaowei Zhang, and Xiao Qin, *Senior Member, IEEE*,

Abstract—Parallel clustering is an important research area of big data analysis. The conventional HAC (Hierarchical Agglomerative Clustering) techniques are inadequate to handle big-scale categorical datasets due to two drawbacks. First, HAC consumes excessive CPU time and memory resources; and second, it is non-trivial to decompose clustering tasks into independent sub-tasks executed in parallel. We solve these two problems by a MapReduce-based hierarchical subspace-clustering algorithm - called *PAPU* - using LSH-based data partitioning. *PAPU* is conducive to partitioning a large-scale dataset into multiple independent sub-datasets, into which similar data objects are mapped. Advocating parallel computing, *PAPU* obtains sub-clusters corresponding to respective attribute subspaces from independent chunks in the local clustering phase. To improve the accuracy of approximated clustering results, *PAPU* measures various scale clusters by applying the hierarchical clustering scheme to iteratively merge sub-clusters during the global clustering phase. We implement *PAPU* on a 24-node Hadoop computing platform. The experimental results reveal that hierarchical subspace-clustering coupled with the data-partitioning strategy achieves high clustering efficiency on both synthetic and real-world large-scale datasets. The experiments also demonstrate that *PAPU* delivers superior performance in terms of extensibility and scalability (e.g., a nearly linear speedup).

Keywords—hierarchical subspace-clustering, LSH-based data partitioning, categorical data, Hadoop

1 Introduction

Hierarchical agglomerative clustering (hereinafter referred to as HAC) is one of the most prominent data analysis techniques thanks to its informative representation of input data's hierarchical structure [1][2]. Despite a handful of advantages, existing HAC algorithms are incapable of dealing with large-scale datasets due to the lack of performance enhancing partitioning [3][4].

In this study, we design a parallel hierarchical subspace-clustering scheme called *PAPU* running on the Hadoop platform. At the heart of PAPU is an efficient data partitioning strategy, which addresses the challenging issue of hierarchical clustering algorithms. We demonstrate how to judiciously partition data among computing nodes to speed up the performance of clustering large-scale categorical data. Our partitioning strategy groups and stores similar data objects in each node in a locality-sensitive hash manner. Similarity measures for categorical data are accomplished by Hamming distance. With our partitioning strategy in place, global hierarchical clustering results can be achieved after performing local subspace clustering on an array of Hadoop nodes in parallel.

1.1 Motivations

The following four observations motivate us to propose PAPU - the Hadoop-based hierarchical subspace-clustering algorithm for categorical data:

- Conventional HAC techniques are insufficient to process big-scale categorical data.
- Appropriate data partitioning is inclined to significantly improve clustering efficiency while maintaining an acceptable accuracy.
- There is a lack of parallel HAC techniques for large-scale categorical datasets.
- The Hadoop platform increasingly becomes a popular practice for processing big data.

Motivation 1. Existing HAC solutions can obtain any number of clusters exhibiting various shapes. An increasing number of applications manage and process large-scale categorical data [5]. Unfortunately, conventional hierarchical clustering schemes (see, for example, [6][7]) are inadequate to handle a massive amount of categorical data.

The existing HAC algorithms are inefficient in clustering big data due to high CPU time and memory complexities. Prior studies show that the traditional clustering methods tend to break down in terms of both accuracy and efficiency during the course of high-dimensional data clustering [8]. In this study, we pay attention to big categorical datasets, because modern real-world categorical datasets are large-scale in nature [9].

Motivation 2. It is futile, if not impossible, to improve parallel clustering efficiency through data partitioning. To make parallel clustering algorithms scalable, one has to enable computing nodes to independently offer local clustering results without collaborating with other nodes. Such an ideal goal can be partially implemented by partitioning data among the computing nodes. It is arguably true that data partitioning plays a key role in optimizing the performance of parallel clustering algorithms.

Taking into account storage locations of data objects, a handful of data partitioning approaches tend to divide

- *N. Pang and J. Zhang* are with Taiyuan University of Science and Technology (TYUST), Taiyuan, Shanxi, China. 030024.*
  *E-Mail: jifuzh@sina.com, *corresponding author: Jifu Zhang.*
- *C. Zhang and X. Qin are with Department of Computer Science and Software Engineering, Samuel Ginn College of Engineering, Auburn University, AL 36849-5347.*
  E-mail:ccz0032@auburn.edu; xqin@auburn.edu.

balanced partitions with either a single hash function or equally spaced keys [10]. Such conventional approaches ignore relevance among data objects, thereby being unable to minimize communication overhead among data nodes [11][12]. Most of the parallel hierarchical clustering methods randomly partition data sets into equal-sized groups prior to clustering. The weakness of such partitions is frequent data transfers among nodes, which pushes I/O overhead to a high level. To address this intriguing problem in data partitioning, we propose a novel solution to group similar objects in each data node in a locality-sensitive manner.

Motivation 3. Although parallel clustering has been intensively investigated, parallel hierarchical clustering is still in its infancy. Developing parallel algorithms for hierarchical clustering is nontrivial, because the HAC schemes have to build a hierarchy of clusters with the dendrogram - a tree structure driven by similarities among all clusters [13].

The HAC algorithms, in each iteration, calculate and store distances among all pairwise clusters. Such pairwise calculations prevent clustering tasks from being decomposed into independent sub-tasks to be executed in parallel. For this reason, implementing parallel solutions for the conventional HAC algorithms is a daunting job. This challenge motivates us to develop a parallel hierarchical clustering algorithm for categorical data. To process large-scale data, we aim to run our parallel hierarchical subspace-clustering algorithm on the Hadoop computing platform.

Motivation 4. Hadoop [14] is a simple yet efficient parallel computing framework offering high scalability and fault tolerance. Very recently, a few inspiring studies demonstrated that Hadoop is a powerful parallel computing technique for clustering massive datasets [15]. In recognizing that little attention has been paid toward hierarchical clustering for big data using Hadoop, we are motivated to design a Hadoop-based clustering algorithm for large-scale categorical datasets. We exploit a way of partitioning data among Hadoop nodes to improve clustering efficiency while maintaining high accuracy.

1.2 Contributions

To facilitate the design of scalable parallel clustering algorithms, we strive to partition a large-scale dataset into multiple independent subdatasets processed by multiple data nodes in parallel. Without such data partitioning, parallel clustering algorithms are slowed down by merging only one pair of closest clusters in each iteration for an entire global dataset.

Our data partitioning relies on the locality sensitive hashing algorithm [16] or *LSH* to project similar data objects into the same bucket. The LSH-based data partitioning improves clustering efficiency by discovering local clusters in each data node without communicating with the other nodes in Hadoop.

After an array of buckets is created using LSH, the buckets may have a diversified number of data objects. To address the data skewness problem imposed by the imbalanced buckets, we design a load balancing index to evenly distribute buckets across Hadoop nodes. The load balancing index's performance largely depends on the bucket granularity. The number of buckets is in a range between the number of Hadoop nodes. We investigate and test the optimal number of buckets to maximize the load balancing performance in the experiment(see Section 7.2).

When a massive amount of datasets are partitioned by the aforementioned LSH, we are in a position to address the challenge imposed by hierarchical clustering. To alleviate high memory demands and long processing time, we implement a two-stage hierarchical subspace-clustering algorithm running on Hadoop. In the first stage, local clustering results are produced corresponding to respective attribute subspaces in parallel from independent chunks split by input datasets. In the second stage, a global dendrogram is formed based on local clustering results of stage one from data nodes using hierarchical agglomerative clustering.

The contributions of this study are summarized as follows:

- We apply locality sensitive hashing to partition data into a set of non-overlapped subdatasets among Hadoop nodes.
- We develop the two-phase - the local and global clustering phase - hierarchical subspace clustering algorithm called *PAPU* running on Hadoop.
- We conduct extensive experiments to evaluate *PAPU* using the *UCI* and *stellar spectral* datasets in addition to synthetic data.

1.3 Organization

The rest of the paper is organized as follows. Section 2 summarizes some preliminaries of this study. The problem statement and main ideas can be found in Section 3. We present a hierarchical subspace-clustering technique using Hadoop computing platform in Section 4. Section 5 discusses the implementation details of *PAPU*. Section 6 and 7 describes the experimental settings as well as the results. Section 8 surveys prior work related to this study. Finally, we conclude our work in Section 9.

2 Preliminaries

In this section, we first introduce hierarchical agglomerative clustering and subspace clustering, followed by an overview of the Hadoop computing framework.

2.1 Hierarchical Agglomerative Clustering

Compared to flat clustering methods, hierarchical clustering algorithms are simple, yet powerful, non-parametric clustering methods. Hierarchical clustering algorithms organize the relationships of clusters using a dendrogram, which shows the positioning course of each data object [17][18]. Hierarchical clustering approaches generally fall into two categories, namely, "top-down" divisive approaches and "bottom-up" agglomerative approaches. Because of adopting an exhaustive search, the time complexity of divisive clustering is $O(2^n)$. In contrast, agglomerative clustering's time complexity is $O(n^3)$, which promotes these types of approaches to be widely adopted. Applying these clustering approaches for processing large-scale data becomes a challenging problem.

Hierarchical agglomerative clustering or *HAC* starts with an initial partition into singleton clusters; HAC iteratively agglomerates an existing closest pair of clusters to create an internal node until all objects gather into the same cluster. At each iteration, all the distances among existing clusters have to be calculated in a distance matrix. The final hierarchical tree structure is called a *dendrogram*, which intuitively shows how clusters are agglomerated at each level.

2.2 Subspace clustering

Low-dimensional adjacent objects within a cluster may be far from each other in a full dimensional space in the context of high-dimensional clustering. The primary cause of such a problem is that high-dimensional objects in different clusters are always correlated with respect to certain attributes subsets. This problem has been addressed by a handful of subspace clustering methods in recent years (see, for example, [19][20]). The overarching goal of subspace clustering is to divide high-dimensional data into multiple low-dimensional subspaces driven by the patterns of the high-dimensional data. The objective of subspace clustering is two-fold, namely, (1) to identify attribute subsets corresponding with clusters and (2) to explore clusters from various attribute subsets.

2.3 MapReduce and Hadoop

Google developed MapReduce [14] as a powerful parallel programming platform to meet the demands of big-data applications. MapReduce simplifies the process of large datasets on parallel computers, where a MapReduce program embraces a pair of mappers and reducers. A mapper is invoked for every record in an input dataset, producing a partitioned and sorted set of intermediate results. A reducer loads the sorted data from an appropriate partition offered by the mapper to compute final output data. Map and reduce functions are conceptually expressed as $map(k1, v1) \rightarrow list(k2, v2)$ and $reduce(k2, list(v2)) \rightarrow (k3, v3)$.

Hadoop is an open-source software implementation of the MapReduce programming model. Data are stored in the Hadoop distributed file system (a.k.a., *HDFS*). Before processing data, files should be imported into HDFS splitting the files into equal sized chunks. The resource management in Hadoop is *YARN*, which splits up the resource management functionalities and job scheduling/monitoring into separate daemons - a global ResourceManager and per-application ApplicationMaster. A Hadoop application is implemented in form of either a single job or a DAG (i.e., directed acyclic graph) of jobs.

3 Proposed Main Ideas

Before presenting the main ideas of *PAPU*, we introduce the problem statement and basic idea of this study below.

3.1 Categorical Data and Problem Statement

Unlike numerical data, categorical data generally refer to particular groups or categories, the values of which are limited, unordered, and non-comparable [21]. An example of categorical data is human blood types, the values of which include A, B, AB, and O. Similarity measures of numerical data using distance are inapplicable for categorical data.

Let $A = \{A_1, A_2, ..., A_d\}$ be a set of d categorical attributes with domain $D(A_1), ..., D(A_d)$, respectively. Domain $D(A_j)$ can be expressed as $D(A_j) = \{D_{j_1}, ..., D_{j_{m_j}}\}$ where mj is the number of categories in attribute A_j $(1 \leq j \leq d)$. We consider a dataset $O = \{O_1, O_2, ..., O_n\}$ of n objects defined on attribute set A, where $O_i = (a_{i1}, a_{i2}, ..., a_{id})$. Object $O_i \in O$ can be represented as a vector $[a_{i1}, ..., a_{id}]$, where a_{ij} $(1 \leq i \leq n, 1 \leq j \leq d)$ is a categorical attribute value in domain $D(A_j)$ (i.e., $a_{ij} \in D(A_j)$). The clustering result can be expressed as a set $C = \{C_1, ..., C_k\}$ where k is the number of clusters C.

In this study, given dataset O, the goal of clustering tasks is to generate tree-like nested partitions represented by $T = \{T_1, ..., T_m\}$. Subcluster C_s is the set of data objects corresponding to the leaves of subtree T_s [22].

Table I lists symbols and notation used throughout this paper.

TABLE I. Symbols and Notations

Symbol	Description	Symbol	Description
O	object set	A	attribute set in O
O_i	the ith object in O	A_j	the jth attribute in A
a_{ij}	attribute value j of object i	$D(A_j)$	domain of attribute j
C_s	the sth cluster	k	the number of clusters T_i
n	the number of objects	d	the number of attributes

3.2 Basic Idea

Prior to the development of *PAPU*, we keep the following two design goals in mind.

- **Design Goal 1.** In the process of constructing a dendrogram, the number of pairwise-distance calculations should be minimized.
- **Design Goal 2.** Parallel clustering algorithms should follow the Hadoop parallel computing model.

To fulfill the above two design goals, we propose a data partitioning strategy as a data reprocessing module in *PAPU*, where a clustering algorithm on Hadoop is in charge of creating a dendrogram in parallel. Traditional hierarchical clustering methods spend an excessive amount of time in comparing and computing pairwise distances from all existing clusters. The data partitioning module in *PAPU* strives to minimize the number of pairwise-distance calculations (see Design Goal 1).

Converting conventional HAC into a parallel algorithm is challenging due to two reasons. First, in each iteration, merging operations depend on global merging results produced from the previous iteration. Second, broadcasting global merging results adversely slows down the overall parallel performance. Our *PAPU* tackles these parallel computing problems by judiciously partitioning data among Hadoop nodes, where local clustering results are computed on Hadoop in parallel during multiple local iterations. To verify the correctness of local clustering results, a reduce phase is to generate how local subclusters from data nodes should be merged according to hierarchical clustering method. (see Design Goal 2).

From the perspective of constructing dendrograms, high-level structures obtained by *PAPU* are almost identical to those built by centralized PAPU and PAPUNSD. Centralized PAPU and PAPUNSD are counterparts to *PAPU* where subspace clustering and data partitioning are intentionally taken out for comparison purpose. The main difference among these three alternative schemes is that during the procedure of building low levels of a dendrogram in each partition, *PAPU* advocates for subspace clustering to produce flat-structure subclusters rather than a subtree.

4 System Design

In this section, we first shed some light on data-partition method (see Sections 4.1). Then, we propose a parallel hierarchical subspace-clustering approach empowered by data partitioning (see Sections 4.2).

4.1 Data Partitioning

Conventional wisdom in data partitioning is to divide input files into logically-independent partitions [23]. Ideally, data-partitioning methods should share the following two hallmarks: (1) pruning redundant objects transmitted among computing nodes; (2) alleviating data skewness among tasks.

Locality sensitive hashing or LSH was proposed to accomplish the nearest neighbour searching in high-dimensional data [24]. We adopt an LSH-based banding technique to address the data-partitioning problem, prior to which characteristic matrix and dimension reduction with MinHash are employed as the theoretical underpinnings of data partitioning.

4.1.1 Dimension Reduction With MinHash

We construct a characteristic matrix, which includes only 1s and 0s, to delineate the attribute space of a dataset. Given dataset O, containing n data objects and m attribute values, we construct an m-by-n characteristic matrix referred to as M, where columns denote data objects and rows mean attribute values. We set the value in position (r, c) to 1 if attribute value r occurs on an attribute of object c; otherwise, the value of (r, c) is configured to 0.

In order to reduce the dimension of the characteristic matrix M, we substitute small-scale representations called "signatures" for the characteristic matrix using the MinHash technique [25]. To generate a signature, we first introduce the minhash function $hmin_j$, which maps object O_i to distinct integers. For any object O_i, we define the minimum value in all hash values as minhash value of $hmin_j(O_i)$. Suppose that the minhash value of object O_i for a certain function $hmin_j$ is denoted by $hmin_j(O_i)$, the signature $Sig(O_i)$ is expressed as $Sig(O_i) = (hmin_1(O_i), hmin_2(O_i), ..., hmin_l(O_i))$.

The basic idea of MinHash is to randomly permute the rows, followed by computing the minhash value (e.g., $hmin_j(c_i)$) of each column (e.g., c_i) to form a signature. A minhash value is the position of the first nonzero attribute-value in the permuted characteristic matrix. Repeatedly performing l independent permutations, we obtain multiple signatures to form an l-by-n signature matrix, where l indicates the number of minhash functions and n signifies the number of objects. It is noteworthy that value l is far smaller than the number of rows m in the characteristic matrix, fulfilling the purpose of the dimension reduction.

4.1.2 Data Partitioning Based on LSH

Locality-Sensitive Hashing or LSH maps similar data objects into same buckets using a banding technique with multiple hash functions [24]. Different from the common hash function, LSH offers a high probability of collision for similar objects. Intuitively, LSH depends on using a large number of hash functions to guarantee that similar data objects are likely to be in the same bucket; however, LSH has a side effect of producing multiple redundant buckets, thereby leading to extra storage overhead [26]. Inspired by the LSH algorithm, we propose a novel data-partitioning method to solve the problem of redundant buckets and bypass computing similarities among a sheer number of object pairs.

Fig. 1 depicts a data-partitioning process employing LSH; the process is comprised of the following steps:

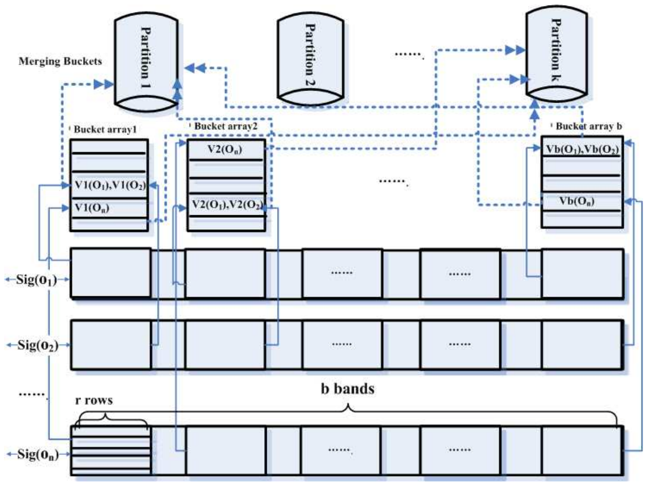


Fig. 1. The LSH-based data partitioning process is comprised of three steps. The first step divides a signature matrix into b bands and maps similar objects into one bucket unit; the second step merges similar bucket units among different bucket arrays; and the third step is responsible for partitioning data to finalize partitioning results.

(1) *Mapping into bucket arrays.* We adopt the traditional LSH technique to obtain the rudimentary bucket arrays. This step is kicked off by dividing the obtained signature matrix into b bands, each of which consists of r rows (see signature $Sig(O_n)$ in Fig. 1). Objects O_i and O_j will be considered alike, if there is at least a corresponding band pair hashing into the same bucket (e.g., O_1, O_2 in Fig. 1). Thus, data objects sharing an identical hash value in the ith band are placed to the unit of the ith bucket array. After repeatedly performing the above operation b times, each data object is successively mapped into b bucket arrays. Finally, a total of $\sum_{i=1}^{b} k_i$ buckets are generated, where b is the number of bucket arrays (see b bucket arrays in Fig. 1). And data object O_i is expressed as a vector $G(O_i) = (v_1(O_i), v_2(O_i), ..., v_b(O_i))$, where v_i represents object O_i's location in ith bucket array.

(2) *Merging similar bucket units.* In the previous step, vector G with b hash values v_i shows that each data object is sequentially mapped into b bucket units from different bucket arrays. Then it shows that similar bucket units are sharing many data objects among the bucket arrays, which give rise to data redundancy and surplus operations. As such, there is a pressing demand to merge similar bucket units using linear hash mapping $H(O_i)$, where vector G of the data object O_i is converted to an integer as the partition's label. Thus, we have $H(O_i) = (a_1 * v_1(O_i) + a_2 * v_2(O_i) + a_d * v_d(O_i)) mod M$, where constant coefficient a_i is in a range between 0 and M–1 (i.e., $a_i \in [0, M–1]$) and M is the number of partitions.

(3) *Partitioning data.* We deploy the results obtained by the above steps to partition objects. The objects belonging to the same bucket hash into the same partition, which can guarantee the objects falling into a partition with great similarity.

4.2 Parallel HAC-Based Subspace

Given input dataset O, hierarchical agglomerative clustering aims to formulate an optimal set C of k clusters $C = \{C_1, \dots, C_k\}$. In light of data partitioning, our *PAPU* algorithm is devised as a two-stage parallel clustering approach that seamlessly integrates the subspace clustering algorithm [27] with the conventional HAC algorithm.

PAPU performs two stages to accomplish parallel clustering tasks. In the first stage of local clustering, *PAPU* utilizes the subspace clustering algorithm in parallel to obtain subclusters in multiple partitions generated by the aforementioned data-partitioning process (see Section 4.2.1). Given subclusters obtained from stage 1, the second stage - referred to as global clustering - employs the hierarchical agglomerative clustering approach to merge the most similar subclusters to achieve global clustering results (see Section 4.2.2).

4.2.1 Local Clustering

To generate local subclusters, we employ a clustering quality function $Q(C)$ to search clustering results using the subspace clustering algorithm [9]. In the local clustering procedure, data objects in different attribute subspace are iteratively scanned to allocate into an existing subcluster or produce a new subcluster to maximize the clustering quality function $Q(C)$.

Quality function $Q(C)$ is expressed as a weighted sum of qualities of items in cluster set C. Thus, $Q(C)$ is measured as follows:

$$\mathrm{Q(C)} = \sum_{\mathrm{s=1}}^{\mathrm{k}} \mathrm{P(C_s)} \times \mathrm{Q(C_s)}, \tag{1}$$

where $Q(C_s)$ (see Eq. 2) denotes the quality of cluster C_s, and $P(C_s)$ is the percentage of objects that belong to cluster C_s. Percentage $P(C_s)$ - serving as a weight - represents the contribution of C_s to overall quality $Q(C)$.

An ideal clustering quality function should make a good tradeoff between compactness and separation [9][28][29].

Here we propose a practical quality function $Q(C_s)$, which combines compactness $Com(a_{ij})$ and separation $Sep(a_{ij})$. Thus, clustering quality $Q(C_s)$ is defined below:

$$\mathrm{Q(C_s)} = \sum_{\mathrm{O_i \in C_s}} \sum_{\mathrm{j=1}}^{\mathrm{d}} [\mathrm{Com(a_{ij})}]^2 \times \mathrm{Sep(a_{ij})} \tag{2}$$

where a_{ij} is an attribute value of data O_i appearing in the jth dimension.

Compactness $Com(a_{ij})$ is measured as

$$\mathrm{Com(a_{ij})} = \frac{\mathrm{count(a_{ij}, A_j, C_s)}}{|\mathrm{O}|} \times \mathrm{W(a_{ij})} = \frac{\mathrm{n_j^s}}{\mathrm{n}} \times \mathrm{W(a_{ij})}, \tag{3}$$

where $count(a_{ij}, A_j, C_s)$ (i.e. n_j^s) is the number of categorical value a_{ij} appearing in dimension A_j of cluster C_s; n is the total number of objects in dataset O. We advocate applying a product of $\frac{n_j^s}{n}$ and weight $W(a_{ij})$ (see Eq. 4) to derive compactness $Com(a_{ij})$, which stresses major contributions of significant attribute values in the clustering formation process.

The subspace clustering principle suggests that attribute weights play a crucial role in data clustering, because an array of attributes is likely to make diversified contributions in clustering of a high-dimensional data. Projecting important attributes on attribute subspaces is a vital step of a subspace clustering analysis.

The significant attribute values composing an attribute subspace not only have a high occurrence frequency in their own dimensions (see Eq. 5), but also have a high co-occurrence degree with the other relevant dimensions of the attribute subspace (see Eq. 6). To address this concern, we assign high weights to such attribute values. We quantify weight of categorical value a_{ij} to precisely assess clustering impacts of attributes. $W(a_{ij})$ is denoted as the weight of attribute value a_{ij}, which is expressed as:

$$\mathrm{W(a_{ij})} = \mathrm{W_{A_j}(a_{ij})} \times [\mathrm{F(a_{ij})}]^2 \tag{4}$$

Weight $W(a_{ij})$ in (4) is a product of single-attribute weight W_{A_j} (see the first term on the right-hand side of (4)) and a_{ij}'s co-occurrence factor F (see the second term on the right-hand side of (4)). To put the spotlight on the impact of co-occurrence frequency, we apply a square to the second term on the right-hand side of (4).

Single-attribute weight $W_{A_j}(a_{ij})$ in (4) represents value a_{ij}'s local clustering capability from the perspective of its own dimension A_j. Weight W_{A_j} can be written as

$$\mathrm{W}_{\mathrm{A_j}}(\mathrm{a_{ij}}) = \frac{\mathrm{n_j}}{\mathrm{n}} \times \log[\mathrm{n_j}(1 - \frac{\mathrm{n_j}}{\mathrm{n}}) + 1]. \qquad (5)$$

where n_j represents the occurrence number of categorical value a_{ij} in attribute A_j, and $\frac{n_j}{n}$ corresponds to the probability of categorical value a_{ij} appearing in dimension A_j.

$$\mathrm{F}(\mathrm{a_{ij}}) = \sum_{\mathrm{s=1, s \neq i}}^{\mathrm{d}} \mathrm{W}_{\mathrm{A_s}}(\mathrm{a_{ij}}) = \sum_{\mathrm{s=1, s \neq i}}^{\mathrm{d}} \frac{\mathrm{n_{js}}}{\mathrm{n_j + n_s - n_{js}}} \qquad (6)$$

where n_s denotes the occurrence number of value a_{is} in dimension A_s. n_{js} counts the number of times that attribute-value pair (a_{ij} and a_{is}) co-occurs in dimensions A_j and A_s. We apply the Jaccard similarity coefficient to calculate the co-occurrence factor of categorical value a_{ij}.

Now we express separation $Sep(a_{ij})$ as follows

$$\mathrm{Sep}(\mathrm{a_{ij}}) = \frac{\mathrm{count}(\mathrm{a_{ij}}, \mathrm{A_j}, \mathrm{C_s})}{\mathrm{count}(\mathrm{a_{ij}}, \mathrm{A_j})} = \frac{\mathrm{n_j^s}}{\mathrm{n_j}}, \qquad (7)$$

where $count(a_{ij}, A_j)$ is the number of categorical value a_{ij} appearing in dimension A_j (i.e. n_j). $\frac{n_j^s}{n_j}$ indicates the percentage of categorical value a_{ij} in dimension A_j exclusively belonging to cluster C_s. A large percentage of a_{ij} implies the importance degree of a_{ij} in the clustering process, which means that a_{ij} plays a vital role in improving the clustering quality of C_s.

4.2.2 Global Clustering

Clustering results yielded by the local-clustering phase merely resemble the distribution of local subclusters from a single computing node. In the course of global clustering, we adopt the HAC algorithm to merge two most similar subclusters with minimum distance iteratively among all subclusters from the different nodes.

There is a large number of linkage measures to quantify the distance (a.k.a., similarity) between clusters; mean linkage averages all distance between pairs of objects from two clusters [30]. The mean linkage measure $dist_{avg}(C_i, C_j)$ is written as:

$$\mathrm{dist_{avg}}(\mathrm{C_i}, \mathrm{C_j}) = \frac{1}{\mathrm{n_i n_j}} \sum_{\mathrm{O_p \in C_i, O_{p'} \in C_j}} |\mathrm{O_p} – \mathrm{O_{p'}}|, \qquad (8)$$

where n_i and n_j express the number of data objects in clusters C_i and C_j, respectively. $|O_p – O_{p'}|$ gauges the distance between two objects $O_p, O_{p'}$. Here, we exploit Jaccard coefficient [31] to assess the distance between two categorical data.

5 Implementation Details

We start this section with an overview of *PAPU* (see Section 5.1). Next, we apply the LSH-based data partitioning method to group similar objects into a single block. The implementation of data-partition can be found in Section 5.2. We also implement a weight-computing module in the Hadoop framework to facilitate subspace clustering, the implementation details of which are well covered in Section 5.3. We discuss the implementation of parallel hierarchical subspace-clustering on Hadoop in Section 5.4.

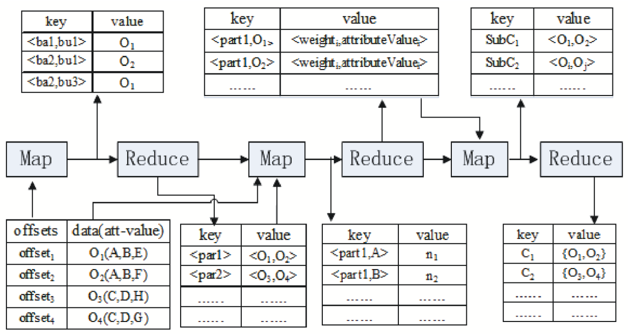


Fig. 2. *PAPU* is comprised of three MapReduce jobs. The first MapReduce job is responsible for partitioning data; the second MapReduce job computes the weight of attribute-values; in the third job, the mapper discovers local subclusters on each data block; and reducer improves the correctness of local subclusters produced by the mapper in the third job through the HAC method.

5.1 Overview

Fig. 2 delineates the working process of *PAPU* consisting of the following three Hadoop jobs seamlessly integrated in *PAPU*.

- **Data Partitioning.** The first job incorporates LSH to map similar objects into a single bucket. We implement the Hadoop-based LSH algorithm to partition data in parallel (see also Section 5.2). We investigate the impacts of LSH-bucket granularity on the efficiency of categorical data clustering. (see the experiment in Section 7.2)
- **Weight Computing.** The objective of the second job is to compute the weight of each attribute-value a_{ij} (see also Section 5.3). Projecting important attributes on attribute subspace is a crucial step of a subspace clustering analysis. We quantify attribute subspace in terms of weight of each attribution value.
- **Parallel HAC on Hadoop.** The clustering process conducted in this job invokes a local clustering module and a global clustering module. Local clustering aims to generate subclusters from similar objects stored on singleton data node. Global clustering generates ultimate dendrogram from all the results obtained from local clustering (see also Section 5.4).

5.2 Data Partitioning

The first MapReduce job in *PAPU* is responsible for partitioning objects of dataset O into multiple data blocks managed by LSH. Algorithm 1 details the pseudocode of the first MapReduce job, which is focused on MapReduce-based data partitioning.

The first job executes the following four steps.

Step 1. Each mapper on a node sequentially reads data objects from a local input split, where each object is stored in the format of pair ⟨LongWritable offset, Text object⟩ (see Line 4).

Step 2. In the mapper's local input split, the mapper sequentially applies two functions (i.e. Generate-chara-matr() and Generate-signature-matr()) to originate the characteristic matrix (see Lines 7-9) and the signature matrix (see Lines 10-12) for each point O_i. The signature matrix

Algorithm 1 Data Partitioning Based on LSH

```
1: input: a dataset;
2: output: distribution of data partitioning;
3: function MAP(key $offset$, values $input$ )
4:     $O' \leftarrow splitter.split(input.toString())$;
5:     $L_{ChaMa} \leftarrow new\ List$;
6:     $L_{Sig} \leftarrow new\ List$;
7:     for all $(O_i : O')$ do
8:         $Append(L_{ChaMa}, Generate\_chara\_matr(O_i))$;
9:     end for
10:    for all $(ch_i : L_{ChaMa})$ do
11:        $Append(L_{Sig}, Generate\_signature\_matr(ch_i))$;
12:    end for
13:    for all $(s_i : L_{Sig})$ do
14:        divide $s_i$ into b bands with r rows;
15:        $Hashbucket \leftarrow HashMap(each\ band\ of\ s_i)$;
16:    end for
17:    applying Linear hash function $H(O_i)$ to compute the partition $P_j$ of each data $O_i$;
18:    emit($P_j$, objectID);
19: end function
20: function REDUCE(key $p_j$,values $objectID$)
21:    $List_p \leftarrow new\ List$;
22:    for all (val: values) do
23:        $Append(List_p, val)$;
24:    end for
25:    emit(partitionID, $List_p$);
26:    Map $A\_list \leftarrow$ all the (partitionID, $List_p$) //$A\_list$ is a CacheFile storing the distribution of data partitioning.
27: end function
```

is generated based on the MinHash algorithm [*]. Lists L_{ChaMa} and L_{Sig} store characteristic matrix and signature matrix of the local input split, respectively.

Step 3. Lines 13-18 perform the implementation of LSH-based data partitioning [†]. LSH divides the signature matrix into b bands, each of which comprises r rows (see Line 14). Subsequently, these bands with same value are mapped into an identical hash bucket that contains similar objects (see Line 15).

As articulated in Section 4.1.2, data object O_i is expressed as a vector composed of d bucket-unit numbers. A Linear hash function $H(O_i)$ is applied to transform a vector to an integer P_j, where O_i is allocated to the partition labeled as P_j. We donate the mapper's output in a tuple format of $\{P_j, objectID\}$ (see Lines 18).

Step 4. A reducer merges all objects with the same partition-label from multiple computing nodes to obtain the complete partition information(see Lines 20-25). The output of the reducer is a list in form of a pair $\{partitionID, List_p\}$ (see Line 25), where $partitionID$ represents the identifier of a partition and $List_p$ expresses a list that contains all objects belonging to partition $partitionID$.

Importantly, a list of pairs in form of $\{partitionID, List_p\}$ (see $A\text{–}List$ in Line 26 of Algorithm1) becomes the second MapReduce job's input, which can be found in Line 3 of Algorithm 3 in the parallel hierarchical subspace-clustering module (see Section 4.2).

*. MinHash code is seen in https://cwiki.apache.org/confluence/display/MAHOUT/Minhash+Clustering

†. LSH code is seen in http://www.mit.edu/ andoni/LSH/

5.3 Weight Computing

The second MapReduce job is in charge of computing attribute-value weights using co-occurrence factors (see Section 4.2.1). It is noteworthy that the input of mappers in this module is the first job's partitioning results. Attribute-value weight is calculated according to distribution of data objects in different data partitions rather than the entire data set.

Algorithm 2 Attribute Weight

```
1: input: $A\_list$;/* $A\_list$ is generated by the first job */
2: output: a weight table of attribute value ;
3: function MAP(key $offset$, values $input$ )
4:     $O \leftarrow splitter.split(input.toString())$;
5:     $L_d \leftarrow newList$;
6:     for all $(O_i : O)$ do
7:         for all $(A_j : A)$ do
8:             for all $(A_s : A)$ do
9:                 $L_d \leftarrow ((a_{ij}a_{is}, j, s), 1)$;
10:            end for
11:        end for
12:    end for
13:    for all $(O_i : O)$ do
14:        for all $(A_j : A)$ do
15:            emit($< a_{ij}, j >, < i, L_d >$);
16:        end for
17:    end for
18: end function
19: function REDUCE(key $< a_{ij}, j >$, values $< i, L_d >$)
20:    $sum \leftarrow newArray$;
21:    for all (val: values) do
22:        $sum[0] \leftarrow sum[0]{+}{+}$; /*$sum[0]$ is the occurrence number of the attribute value $a_{ij}$ on a single dimension $A_j$*/
23:        for all (s=1;s<=j;s++) do
24:            if $(a_{ij}a_{is})$ then
25:                $sum[s] \leftarrow sum[s]{+}{+}$; /*$sum[s]$ means the number of attribute value pair co-occurring on dimension $A_j$ and $A_s$*/
26:            end if
27:        end for
28:    end for
29:    computing $W_{A_j}(a_{ij})$ with formula (5) and $sum[s]$;
30:    emit($< a_{ij}, i, j >$, $W_{A_j}(a_{ij})$) and Map $B\_list \leftarrow$ all the ($< a_{ij}, i, j >, W_{A_j}(a_{ij})$);
31: end function
```

Algorithm 2 depicts the pseudocode of the second job, which carries out the following four steps.

Step 1. Data-partition set L_A is derived from list A_list (see Line 26 in Algorithm 1), which is an input data set of the mapper (see Line 4 in Algorithm 2).

Step 2. To quantify the weight of attribute value a_{ij}, the mapper counts (1) the occurrence of each attribute value in a single dimension and (2) the co-occurrence of each attribute-value pair among multiple dimensions (see Lines 4-17). ListL_d keeps track of the co-occurrences among the multiple dimensions.

Step 3. To alleviate high network communication overhead experienced in the shuffle stage, we aggregate a large number of small key/value pairs into a large compound one (see Lines 13-17). Tuple $< a_{ij}, j >$ is taken as the key of mapper's output, where j is an index of dimension on which attribute value a_{ij} is located. The compound value of mapper's output is tuple $< i, L_d >$, where the elements of list L_d are tuple $< (a_{ij}a_{is}, j, s), 1 >$. $a_{ij}a_{is}$ is the attribute-value pair $\{a_{ij}, a_{is}\}$ co-occurring on dimensions A_j and A_s (see in Line 9).

Step 4. A reducer merges local occurrences of attribute values and co-occurrences of attribute-value pairs to compute the weight value of a_{ij} (see Lines 19-31). The output of the reducer is a list in the format of pair $\langle < a_{ij}, i, j >, W(a_{ij}) \rangle$ (see Line 30). The output of the second job is a list (i.e., B-list) of pairs in the format of $\{< a_{ij}, i, j >, W(a_{ij})\}$, which is fed into the third MapReduce jobs as an input (see Line 3 of Algorithm 3).

5.4 Parallel Clustering with Hadoop

With data partitioning and weight computing in place, we implement a two-step approach to obtaining hierarchical subspace-clustering results in the third MapReduce job. It is worth noting that the input of mappers in the third job is the output of the reducers in the second job. In step one, adopting the subspace clustering method (see Section 4.2.1), local clustering results are produced by the third job's mappers (see the mapper of Algorithm 3) on each data node. When it comes to the second step, the global clustering stage aggregates local clusters to determine a dendrogram using the hierarchical agglomerative clustering algorithm (see the reducer of Algorithm 3).

5.4.1 *Local-step-based Subspace Clustering*

The local-clustering stage performs the following four steps to form local subclusters of different partitions in the mapper, the output of which is a tuple of $\langle subclusterID, sc_i \rangle$. $subclusterID$ in the tuple is an identification of subcluster sc_i. Subcluster sc_i is yielded based on subspace clustering algorithm (see Section 4.2.1). These pairs generated by the third job's mappers are shuffled and combined into the reducers, which are technical underpinnings of the global-clustering stage.

Step 1. A table (see Line 1 in Algorithm 3) of weighted partitions is created by loading list $B\text{–}list.key$, where $B\text{–}list$ is an output of the second job (see Line 30 in Algorithm 2).

Step 2. From a local input split, each mapper sequentially reads data objects and randomly selects a data object to store in list SC as the first subcluster (see Line 5). Here, list SC stores subclusters corresponding to local input splits.

Step 3. This process is outlined in Lines 7-26 of Algorithm 3. To maximize the clustering-quality function $Q(C)$ (see Eq.1), we address two vital issues, namely, (1) each data object (e.g., O_i) of the input split is assigned to an existing subcluster or creates a new subcluster to maximize $Q(C)$ in initial clustering (see Lines 7-21), and (2) some subclusters are merged to maximize $Q(C)$ in the local-merging stage (see Lines 22-26), thereby guaranteeing accuracy of initial clustering results in addition to decreasing communication overhead.

Step 4. The output of the mapper is a list of pairs $\{subclusterID, sc_i\}$, in which subcluster sc_i produced by the preceding steps indicates local subspace-clustering results (see also Lines 27-29).

Algorithm 3 Parallel Hierarchical Subspace-Clustering.

```
1: input: dataset, $B\text{–}list$; ;/* $B\text{–}list$ is generated by the second job */
2: output: clustering result;
3: function MAP(key $offset$, values $input$ )
4:     $O \leftarrow splitter.split(input)$;
5:     $sc_l \leftarrow$ any one data object of $O$;
6:     $max \leftarrow 0$;
7:     for all  $(O_i : O)$  do
8:         for all  $(sc_k : SC)$  do/*$SC$ is the subcluster set */
9:             if (Q($\{O_i\} \cup sc_k$)-Q($\{O_i\}$)-Q($sc_k$)>0)  then
10:                if (Q($\{O_i\} \cup sc_k$)> max) then/* Q() is cluster quality computed using Eq. 1*/
11:                    maxid=k;
12:                    max=Q($\{O_i\} \cup sc_k$);
13:                end if
14:            else
15:                $SC \leftarrow \{O_i\}$; /*$O_i$ generates a new subcluster*/
16:            end if
17:        end for
18:        if $(max! = 0)$ then
19:            $SC \leftarrow sc_{maxid} \cup \{O_i\}$; /*$O_i$ is allocated a subcluster*/
20:        end if
21:    end for
22:    for all  $(sc_i : SC)$ do
23:        for all  $(sc_j : SC)$ do
24:            Merge subcluster $sc_i$ and $sc_j$ to maximize $Q(SC)$;
25:        end for
26:    end for
27:    for all  $(sc_i : SC)$  do
28:        emit($subclusterID$,$sc_i$);
29:    end for
30: end function
31: function REDUCE(key $subclusterID$, values $sc_i$)
32:    for all  $(val : values)$ do
33:        $SC \leftarrow val$;
34:    end for
35:    for all  $(sc_i : SC)$  do
36:        for all  $(sc_j : SC)$  do
37:            search most similar two subcluster ($sc_i$ and $sc_j$);
38:            Merge $sc_i$ and $sc_j$ to generate a dendrogram with HAC method;
39:        end for
40:    end for
41:    emit($key$,$dendrogram$);
42: end function
```

5.4.2 *Global-step-based HAC*

Local clustering results yielded by the aforementioned mappers may be insufficiently accurate from a global perspective. This problem is solved by global clustering handled in the reducers of Algorithm 3, which executes two steps to originate a hierarchical structure of global clustering results. The input of a reducer is a list of pairs (i.e., $\{subclusterID, sc\}$) obtained from the third job's mapper output (see Line 28 in Algorithm 3).

Step 1. SC is created to maintain ultimate clustering results in the form of a dendrogram (see Lines 32-34 in Algorithm 3).

Step 2. A global clustering result is made possible by iteratively aggregating the most similar subclusters using the hierarchical agglomerative clustering algorithm (see Section 2.1). Each reducer emits a pair $\langle key, dendrogram \rangle$, where $dendrogram$ means the global clustering result in format as hierarchical tree structure (see Lines 35-42).

6 Experimental Setup

We evaluate the performance of *PAPU* on a real-world 24-node Hadoop cluster. Each computing node has an Intel E5-1620 v2 series 3.7G quad core processor, 16 GB main memory. We install the Centos 6.4 OS, Java JDK 10.0.2, and Hadoop 3.0 on the cluster. The capacity of disks in the namenode and datanode are 500GB and 2TB, respectively. We use the default parameters to configure the Hadoop cluster.

6.1 Datasets

The performance evaluation is driven by both synthetic (see Section 6.1.1) and real-world (see Section 6.1.2) datasets processed by *PAPU* on the Hadoop cluster.

TABLE II. Characteristics of the synthetic and real-world datasets.

Dataset	Type	Size(MB)	# Objects	# Dims	# Clusters
Type1	Synthetic	2×10^3	0.5×10^8	100	20
Type2	Synthetic	4×10^3	1×10^8	100	20
Type3	Synthetic	6×10^3	2×10^8	100	30
Type4	Synthetic	8×10^3	4×10^8	100	30
Splice	Real(UCI)	0.312	3190	61	3
Mushroom	Real(UCI)	0.365	8124	22	2
Spectrum	Real	349	520403	91	5

6.1.1 Synthetic Dataset

The four synthetic datasets (i.e., $Type1 \sim Type4$) are generated to resemble various practical applications reported in the literature [27]. We create the four synthetic datasets using an existing data generator toolkit [32], which is available on the web site‡.

The structural discrepancies among the four synthetic datasets are vividly demonstrated in Fig. 3. The attribute subspaces are completely disjoint among multiple clusters in $Type1$. $Type2$ has an array of sharing objects across a group of clusters. There is intersection among the attribute subspace of different clusters in $Type3$. Unlike $Type1$, $Type4$ is a non-equilibrium dataset, cluster sizes of which change greatly.

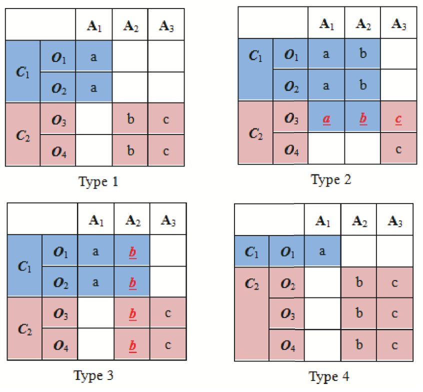


Fig. 3. $Type1 \sim Type4$ are four synthetic datasets. In these datasets, there are three categorical attributes (i.e., A_1, A_2, A_3), four objects (i.e., O_1, O_2, O_3, O_4), and two clusters (i.e., C_1, C_2).

‡. http://www.datgen.com

6.1.2 Real-World Datasets

Two real-world datasets (see also Table II) are processed to test the clustering algorithms. The first dataset is selected from the widely adopted UCI repository§(i.e. Splice [27], Mushroom [29][28][9][32]); the second one is the Celestial Spectrum data released by the China National Astronomical Observatory [33]. All the continuous dimensions are discretized into finite categorical values in the Spectrum datasets.

6.2 Performance Metrics

The three evaluation criteria or metrics applied to assess the quality of clustering algorithms include (1) *adjusted rand index* (ARI), (2) Jaccard index ($Jaccard$), and (3) purity metric (Pur) [9].

Adjusted rand index or ARI mainly evaluates the similarity between clustering results and actual clusters (a.k.a., ground-truth clusters). $Jaccard$ determines the proportion of correctly divided data pair to total ones. Pur expresses the ratio of dominant data objects in each cluster; Pur is calculated as a weighted sum of each cluster's purity. A large value indicates good lustering performance for the three performance metrics (ARI, $Jaccard$ and Pur).

We employ the Speedup measures [34] to quantify the parallel performance of our *PAPU*; the Speedup metric is calculated as follows:

$$\mathrm{Speedup(p)} = \frac{\mathrm{T_1}}{\mathrm{T_p}} \tag{9}$$

where p represents the number of nodes, T_1 is the execution time of the parallel algorithm running on a single processor and T_p is the execution time spent on p nodes processing the same amount of data. The speedup metric intuitively shows the scalability of the parallel algorithm.

7 Experimental Results

7.1 Data-partitioning Impacts on Clustering

In the first group of experiments, we evaluate the impacts of data partitioning on clustering efficiency of *PAPU* driven by an assortment of synthetic datasets. Fig. 4 shows the efficiency of the data-partitioning-enabled and non-data-partitioning-enabled *PAPU* running on the 16 computing nodes.

Let us vary the dataset type from $type1$ to $type4$, the running times of which are plotted in Fig. 4. The average running time of the data-partitioning-enabled *PAPU* is 18.25% faster than that of the non-data-partitioning-enabled counterpart in the tested datasets. The reason is two-fold. First, the data-partitioning strategy maps similar data into one block prior to the clustering phase, based on which *PAPU* effectively reduces clustering ranges while minimizing the number of pairwise-distance computations. Second in the local-clustering phase of the data-partitioning-enabled *PAPU*, the number of subclusters in the each mapper is significantly decreased, thereby effectively reducing the reduce-phase running time.

§. http://archive.ics.uci.edu/ml/datasets.html

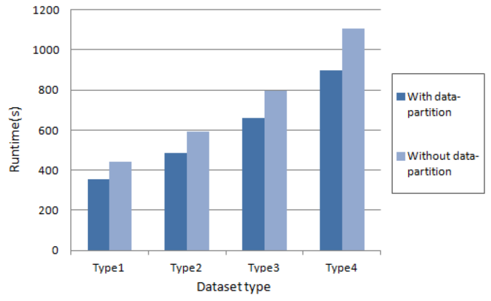


Fig. 4. Clustering-efficiency comparisons between the data-partitioning-enabled *PAPU* and the non-data-partitioning-enabled *PAPU*. We vary the data distribution and the number of data objects in the four testing datasets. The number of computing nodes is set to 8.

7.2 Impact of LSH-bucket Granularity

Locality Sensitive Hashing or LSH is applied to produce multiple buckets in our data-partition strategy (see Section 4.1). In this group of experiments, we evaluate the impacts of bucket granularity on the efficiency of *PAPU*, where we employ the ratio of bucket number to computing node number to quantify bucket granularity. The number of buckets - representing the number of map tasks - theoretically implies the degree of parallelism.

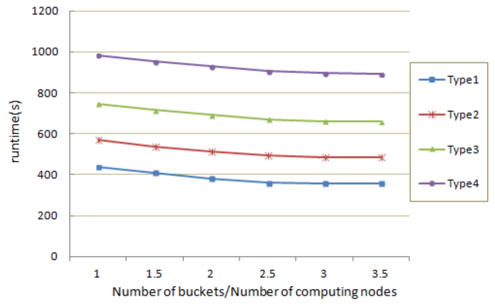


Fig. 5. Impacts of bucket quantity on the efficiency of PAPU. Four synthesis datasets are tested (i.e., Type1, Type2, Type3 and Type4). The number of computing nodes is set to 8.

Fig. 5 reveals that the running time of *PAPU* drops with the increasing bucket granularity. Interestingly, the reduction trend diminishes gradually thanks to three main reasons. First, the total execution time of the third job is determined by the slowest worker node and incompletely rest with the number of map tasks. Second, increasing the number of buckets consumes more time in merging subclusters yielded during the course of the local clustering phase. The efficiency improvement stops at a particular number of bucket granularity due to detrimental I/O overheads. Finally, the entire running time is constrained by the global clustering step, which leaves little room for improvement contributed by the sequential clustering process with an increasing number of map tasks.

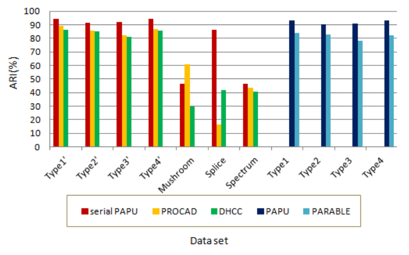
7.3 Comparison s with Existing Algorithms

The goal of this group of experiments is to assess the clustering accuracy and efficiency of our *PAPU* and the other alternative clustering algorithms (see Table III).

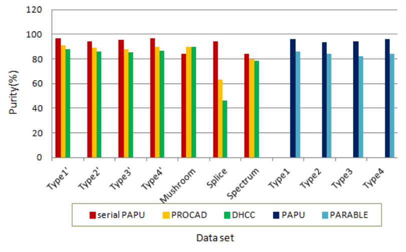
We contrast the clustering precision and efficiency of our *PAPU* with DHCC [28], PROCAD [9] and PARABLE [15] driven by the seven testing datasets, which are composed of Splice and Mushroom available from the UCI datasets and the four synthetic datasets and the subsets extracted from the spectrum dataset. DHCC is a typical non-parametric clustering algorithm using the idea of HAC for reference. PROCAD is a subspace clustering algorithm based on attribute-weight calculation using frequency of attribute values in a single dimension. DHCC and PROCAD are sequential clustering algorithms running a single node. To fairly compare with DHCC and PROCAD, we run our PAPU without data-partition on a single node(a.k.a.,serial PAPU). We acquire the partial source code of the DHCC algorithm from its author. We also realize PROCAD and DHCC use the programming language *Java* and the integrated development environment *Eclipse*. We implement the parallel clustering algorithm (i.e., PARABLE) using Java JDK 10.0.2 and Hadoop 3.0. PARABLE is the epitome of parallel hierarchical clustering algorithms that take full advantage of random data partitioning (see Section 8 for the details). The number of partitions M as the parameter of PARABLE is set to 30.

We observe from Fig. 6 that the differences among the three competitive algorithms are insignificant in the four synthetic datasets. Fig. 6 shows that the clustering-evaluation indicator of *serial PAPU* is superior to those of the other two algorithms in most of the testing datasets; this trend is especially true in the Splice dataset. The attribute values of the Splice dataset have only four options (i.e., A, T, G, C). A small domain usually leads to low discrimination of attribute weight, thereby affecting final clustering results of the clustering algorithms that rely on a single attribute (e.g., PROCAD). *Serial PAPU* obtains an optimal clustering effect in the Splice dataset, because subspace clustering powered by the co-occurrence frequency of attribute value (see Section 4.2.1 for details) is accomplished in dealing with the dataset consisting of the small-range attributes in the local clustering phase. The clustering performance of *serial PAPU* on Mushroom is second only to PROCAD. The key reason is that *serial PAPU* benefits from the multi-attribute frequency of attribute weights using Eq. 4 at the cost of meticulous clustering problems. As such, *serial PAPU* ensures the purity of subclusters while affecting the other two indicator measures.

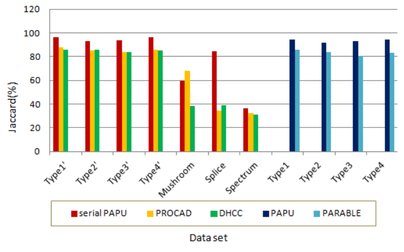
Fig. 6 shows that our *PAPU* is superior to PARABLE in terms of clustering precision. In the map phase, the PARABLE algorithm randomly divides an entire dataset into an array of smaller partitions, each of which is handled by the sequential HAC algorithm to form local clustering results in mappers. There is no information exchange among the mappers. Using the LSH-based data-partitioning strategy, our *PAPU* is adept at grouping similar data objects into one data block or more prior to data clustering. *PAPU* delivers higher accuracy than PARABLE, because *PAPU* orchestrates subspace clustering to yield multiple subclusters that lead to an approximate global dendrogram.



(a) Compare PAPU with the existing algorithms in terms of ARI.



(b) Compare PAPU with the existing algorithms in terms of Purity.

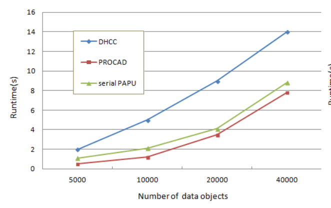


(c) Compare PAPU with the existing algorithms in terms of Jaccard.

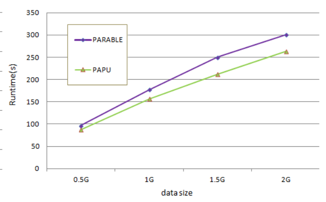
Fig. 6. Compare PAPU with the existing algorithms in terms of ARI, $Jaccard$, and Pur. The seven tested datasets include the UCI dataset Mushroom, Splice and subsets extracted from the synthetic, spectrum dataset. The three existing algorithms are DHCC [28], PROCAD [9] and PARABLE [15].

Fig. 7(a) reveals clustering efficiency of sequential PAPU and the other two sequential algorithms. We vary the number of data objects (i.e., from 5000 to 40000) extracted from synthetic dataset Type1. The extensibility of sequential *PAPU* is only second to PROCAD. *PAPU* calculates attribute weights according to co-occurrence frequencies among multiple attributes. With a growing number of data objects, *PAPU*'s computation time surges. Fortunately, our *PAPU* merely searches, in the subspace-clustering phase, the most appropriate one among the existing subclusters. In other words, *PAPU* avoids searching an entire dataset to slash computing cost. *PAPU*'s time spent in forming subclusters is significantly lower than DHCC's time consumed in merging one pair of closest clusters.

Fig. 7(b) plots an efficiency trend of *PAPU* and PARABLE when the data size varies from 0.5 GB to 2.0 GB. *PAPU* outperforms PARABLE thanks to two reasons. First, *PAPU* adopts the based-LSH data partitioning method to split big data into small and independent partitions, which helps in effectively reducing network load and cutting redundant data exchange. Second, in each computing node, subspace clustering on similar objects can dramatically improve the clustering speed as well as parallelism.



(a) Compare serial PAPU with DHCC and PROCAD varying the number of data objects.



(b) Compare PAPU with PARABLE varying the number of data objects.

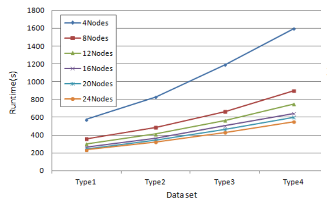
Fig. 7. Clustering-efficiency comparisons between PAPU and the existing algorithms. We vary the number of data objects in synthetic dataset Type1.

7.4 Extensibility

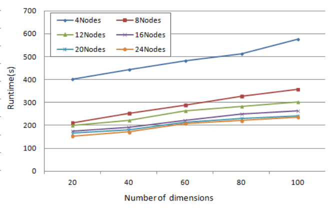
In this group of experiments, we mainly evaluate the extensibility of our *PAPU* on the Hadoop platform. Fig. 8 plots the running times of *PAPU* as a function of the number of objects and attributes, respectively.

Fig. 8(a) shows the impacts of data size (i.e., varying from 2.0 GB to 8.0 GB) and the number of computing nodes on *PAPU*'s running time. We observe that expanding the data size gives rise to the increasing execution time in almost a linear manner. Fig. 8(a) clearly reveals that the overall running time of *PAPU* is enlarged along with an increasing dataset size. The large amount of time spent in dealing with a large-scale dataset is obviously alleviated by increasing computing nodes.

Fig. 8(b) depicts the execution times of *PAPU* when the number of dimensions varies from 20 to 100. Adopting co-occurrence frequencies among multiple dimensions in the subspace-clustering stage of *PAPU*, we observe that the time spent in each mapper rises when the number of dimensions goes up. Fig. 8(b) shows when we expand the number of computing nodes from four to 24, the execution-time increasing ratio is reduced by a factor of 2.11. Intuitively, scaling up the Hadoop cluster size (i.e., the number of computing nodes) is conducive to the acceleration of the subspace- clustering process. Nevertheless, the *PAPU*'s performance difference between the 20-node and 24-node cases is marginal due to adverse communication overhead among a large number of nodes.



(a) Impacts of the numbers of objects on *PAPU*'s running times measured in seconds.

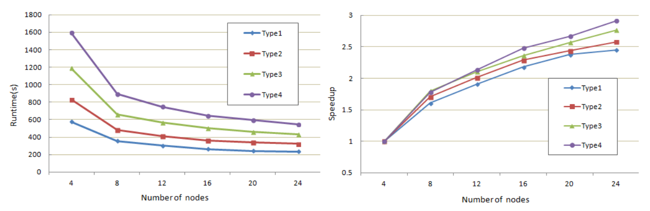


(b) Impacts of the numbers of attributes on *PAPU*'s running times measured in seconds.

Fig. 8. The extensibility of the *PAPU* algorithm. The number of computing nodes is varied from 4 to 24. Tested datasets include $Type1$, $Type2$, $Type3$, and $Type4$, the data size of which varies from 2.0 to 8.0 GB. The attributes size is configured in a range between 20 and 100.

7.5 Scalability

The scalability of *PAPU* is evaluated by increasing the number of computing nodes in Hadoop platform from 4 to 24 with an increment of 4.



(a) PAPU's running times of processing the four datasets

(b) PAPU's speedups of processing the four datasets

Fig. 9. The scalability of the *PAPU* algorithm. Tested datasets include the four synthetic datasets. The data size of $Type4$ is the largest and that of $Type1$ is the smallest. The number of computing nodes is varied from 4 to 24.

The results plotted in Fig. 9(a) indicate a significant downtrend for the running time of *PAPU* with the expanding number of computing nodes. Such a trend becomes more pronounced for large-scale datasets. The results apparently confirm that *PAPU* is a parallel hierarchical subspace-clustering algorithm delivering good scalability regardless of the synthetic or real-world datasets. We ascribe the high scalability of *PAPU* to three factors. First, the data-partitioning strategy divides a large dataset into a number of data blocks uniformly stored on multiple nodes. Running time of *PAPU* is inversely proportional to the number of computing nodes. Second, counting co-occurrence frequencies among relevant dimensions is a very time-consuming yet important stage, the execution time of which is proportional to the number of data objects assigned to one node. Third, *PAPU*'s local-clustering phase performs parallel computing, where subsets of the data partitions are handled concurrently.

Fig. 9(b) demonstrates that *PAPU* exhibits an approximately linear speedup ratio indicating high parallel efficiency on most of the datasets, particularly the big ones. The rationale behind good speedup performance is given as follows. When it comes to large datasets, *PAPU* introduces fairly low interconnect-communication overhead compared with heavy computing load. In the small-data cases, high communication overhead overshadows light computing workload. Thus, it is arguably true for small datasets that an optimum point of PAPU's scalability is likely to occur with an increasing number of nodes. *PAPU*'s communication overhead incurred by computing global dendrograms simply accounts for a small portion of the overall running time. The main reason is two-fold. First, thanks to LSH-based data partitioning, most data objects are assigned to their final location of dendrogram in the map stage. Second, the shuffle stage is responsible for transferring intermediate data from mappers to reducers. Sorting is an important yet time-consuming component in the shuffle process. A salient strength of LSH-based data-partitioning is that a significant portion of intermediate data are pre-sorted in the *PAPU*'s map phase, thereby making merge-sorting operation trivial in the reduce phase.

8 Related Work

In this section, we first review existing techniques in the realm of data-correlation-based data partitioning (see Section 8.1). Then, we discuss parallel hierarchical clustering schemes or parallel HAC in Section 8.2. Table III summarizes the comparisons between *PAPU* and the existing clustering schemes.

TABLE III. Comparisons among the sequential and parallel clustering algorithms.

Algorithms	Platform	Partitioning	Subspace	HAC	Data
PROCAD[9]	NO	×	✓	×	C
DHCC [28]	NO	×	×	✓	C
Li [35]	SIMD	×	×	✓	N
Olson[36]	PRAM	×	×	✓	N
HybridHC[2]	OpenMP	✓	×	✓	S
pPOP[4]	MPI	✓	×	✓	N
Z. Du[37]	MPI	×	×	✓	N
PARABLE [15]	Hadoop	×	×	✓	N
IncDiSC[38]	Hadoop	×	×	✓	I
PAPU	Hadoop	✓	✓	✓	C

8.1 Data-Partition Method

Data partitioning in database systems has been widely studied in both single-server systems and distributed systems [39]. Data-partitioning techniques have been applied in a variety of fields such as frequent subgraph mining [40], kNN joins [41], and frequent itemset mining [39]. A recent study suggests that data-partitioning methods play a vital role in parallel hierarchical clustering [2][13][42]. An efficient data-partitioning scheme can reduce the number of pairwise distances computed for hierarchical clustering. More importantly, data-partitioning methods contribute to hierarchical-clustering results with performance guarantees. Data partitioning strategies that rely on data correlations offer an effective way of solving large-scale clustering problems using hierarchical clustering algorithms. In short, the efficiency of the parallel HAC algorithms is improved using a valid partitioning scheme without compromising their accuracy [4].

An adaptive Landmark-based AHDC [43] method was proposed to partition a large-scale sequence dataset into groups [2]. As a building data-block technique, the adaptive landmark-based selection scheme was applicable for flat clustering. Although coming with theoretical guarantees on clustering performance [43], two major drawbacks, relatively high time complexity and poor noise immunity, make AHDC inadequate for dealing with large datasets.

Using partially overlapping partitioning, Dash *et al.* [4] proposed the *pPOP* algorithm converting a big-data clustering problem into multiple local-clustering problems. A dataset is partitioned into c overlapping cells with parallel axis. In each iteration, the *pPOP* algorithm searches the closest pair of clusters for each cell, followed by determining the overall closest pair from those pairs. Only one pair of clusters can be selected as the result of each iteration.

8.2 Parallel Hierarchical Clustering

Compared to the flat clustering methods, hierarchical clustering algorithms offer several advantages thanks to their non-parametric nature and the ability to elucidate the overall structure of a dataset. A nonparametric algorithm *PROCAD* was presented by Bouguessa [9] as a project clustering method extended from hierarchical clustering. PROCAD evaluates the clustering weights of attributes with occurrence frequency on their own dimensions. Xiong *et al.* [28] proposed *DHCC* - a hierarchical clustering scheme. *DHCC* splits an initial cluster based on a multiple correspondence analysis, which repeatedly chooses one cluster to split into two subclusters optimizing clustering performance.

Although hierarchical clustering offers a handful of advantages, such strategy faces a challenge of being effectively parallelized due to high computational cost and data dependences. Consequently, little work has been carried out on parallel hierarchical clustering [2][15][4][1], especially in the context of the Hadoop framework [15].

Du *et al.* developed a parallel hierarchical clustering algorithm using distributed memory architectures [37]. This algorithm, however, is unadaptable to process large datasets due to a prerequisite of calculating a distributed distance matrix.

In the aforementioned hierarchical clustering approaches, it is an open issue to create global clustering results by generating local sub-clusters constructed on each computing node. Conventional parallel hierarchical clustering algorithms improve efficiency by distributing iterative-agglomerating tasks into multiple computing nodes. Unfortunately, random data partitioning adopted by traditional parallel hierarchical clustering are inadequate for achieving both high accuracy and efficiency. It is worth noting that such a local-result merging process is a key to clustering accuracy. More importantly, one major discrepancy between the above algorithms and our *PAPU* is that our scheme leverages local results to approximate global dendrogram thanks to data partitioning.

9 Conclusion and Future Work

In this paper, we investigated the problem of hierarchical clustering for categorical data. We paid particular attention on parallel hierarchical clustering using the MapReduce programming model. Our approach addresses the two problems encountered in the existing hierarchical clustering techniques. First, due to high computational cost and data dependencies, hierarchical clustering is difficult to parallelize. Second, random partitioning methods running on the MapReduce computing framework are inadequate for significantly improving clustering efficiency and maintaining an acceptable accuracy.

We proposed a MapReduce-based hierarchical subspace-clustering algorithm *PAPU* coupled with attribute-value weights using a data-partitioning strategy. Conducting extensive experiments driven by the synthetic and real-world large-scale datasets, we compared our *PAPU* with two existing clustering algorithms - PROCAD and DHCC. The experimental results show that the *PAPU* scheme outperforms the two existing algorithms in terms of adjusted rand index, Jaccard index, and the purity metric in most of the testing dataset. Furthermore, the results also demonstrate that running on the MapReduce framework, *PAPU* achieves high performance in terms of extensibility and scalability. Importantly, *PAPU* exhibits an approximately linear speedup with the number of computing node in a Hadoop cluster.

As a future research direction, we intend to adopt an adjustable mechanism based on multi-granularity databucket to solve the data skewness problem. Such a mechanism aims to judiciously assign buckets with diversified granularities to computing nodes by taking available resources into account. We will employ this mechanism to mitigate data skewness in a follow-up study.

Acknowledgment

This work is supported by the National Natural Science Foundation of China under Grant No.61876122. Xiao Qin's work is supported by the U.S. National Science Foundation under Grants IIS-1618669, CNS-0917137, and CCF-0845257. The authors would like to thank Mojen Lau and Christian Kauten for proofreading and editing this paper.

References

[1] Y. Jeon and S. Yoon, "Multi-threaded hierarchical clustering by parallel nearest-neighbor chaining," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 9, pp. 2534–2548, 2015.

[2] Q. Mao, W. Zheng, L. Wang, Y. Cai, V. Mai, and Y. Sun, "Parallel hierarchical clustering in linearithmic time for large-scale sequence analysis," in *IEEE International Conference on Data Mining*, 2016, pp. 310–319.

[3] S. Rajasekaran, "Efficient parallel hierarchical clustering algorithms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 6, pp. 497–502, 2005.

[4] M. Dash, S. Petrutiu, and P. Scheuermann, "ppop: Fast yet accurate parallel hierarchical clustering using partitioning," *Data and Knowledge Engineering*, vol. 61, no. 3, pp. 563–578, 2007.

[5] C. Keribin, V. Brault, G. Celeux, and G. Govaert, "Estimation and selection for the latent block model on categorical data," *Statistics and Computing*, vol. 25, no. 6, pp. 1201–1216, 2015.

[6] J. Wang, M. Li, J. Chen, and Y. Pan, "A fast hierarchical clustering algorithm for functional modules discovery in protein interaction networks," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 8, no. 3, pp. 607–620, 2011.

[7] D. Wei, Q. Jiang, Y. Wei, and S. Wang, "A novel hierarchical clustering algorithm for gene sequences," *BMC Bioinformatics*, vol. 13, no. 1, pp. 1–15, 2012.

[8] A. A. Esmin, R. A. Coelho, and S. Matwin, "A review on particle swarm optimization algorithm and its variants to clustering high-dimensional data," *Artificial Intelligence Review*, vol. 44, no. 1, pp. 23–45, 2015.

[9] M. Bouguessa, "Clustering categorical data in projected spaces," *Data Mining and Knowledge Discovery*, vol. 29, no. 1, pp. 3–38, 2015.

[10] N. Benjamas and P. Uthayopas, "Impact of i/o and execution scheduling strategies on large scale parallel data mining," in *Information Science and Service Science and Data Mining*, 2013, pp. 654–660.

[11] J. M. Cope, N. Trebon, H. M. Tufo, and P. Beckman, "Robust data placement in urgent computing environments," in *IEEE International Symposium on Parallel and distributed Processing*, 2009, pp. 1–13.

[12] T. Xie, "Sea: A striping-based energy-aware strategy for data placement in raid-structured storage systems," *IEEE Transactions on Computers*, vol. 57, no. 6, pp. 748–761, 2008.

[13] C. Jin, M. M. A. Patwary, W. Hendrix, A. Agrawal, W. K. Liao, and A. Choudhary, "Disc: A distributed single-linkage hierarchical clustering algorithm using mapreduce," *Work*, 2013.

[14] Z. Liu, Q. Zhang, R. Ahmed, R. Boutaba, Y. Liu, and Z. Gong, "Dynamic resource allocation for mapreduce with partitioning skew," *IEEE Transactions on Computers*, vol. 65, no. 11, pp. 3304–3317, 2016.

[15] S. Wang and H. Dutta, "Parable: A parallelrandom-partition based hierarchicalclustering algorithm for the mapreduce framework," *Center for Computational Learning Systems Columbia University*, 2011.

[16] J. Qian, Q. Zhu, and H. Chen, "Multi-granularity locality-sensitive bloom filter," *IEEE Transactions on Computers*, vol. 64, no. 12, pp. 3500–3514, 2015.

[17] A. Bouguettaya, Q. Yu, X. Liu, X. Zhou, and A. Song, "Efficient agglomerative hierarchical clustering," *Expert Systems with Applications*, vol. 42, no. 5, pp. 2785–2797, 2015.

[18] M. Ackerman and S. Ben-David, *A characterization of linkage-based hierarchical clustering*. JMLR.org, 2016.

[19] A. Adler, M. Elad, and Y. Hel-Or, "Linear-time subspace clustering via bipartite graph modeling," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 26, no. 10, pp. 2234–2246, 2015.

[20] G. Gan and K. P. Ng, "Subspace clustering using affinity propagation," *Pattern Recognition*, vol. 48, no. 4, pp. 1455–1464, 2015.

[21] T. D. Wickens, "Categorical data analysis." *Annual Review of Psychology*, vol. 49, no. 1, pp. 109–109, 1998.

[22] K. A. Heller and Z. Ghahramani, "Bayesian hierarchical clustering," in *International Conference on Machine Learning*, 2005, pp. 297–304.

[23] A. K. Paul, W. Zhuang, L. Xu, M. Li, M. M. Rafique, and A. R. Butt, "Chopper: Optimizing data partitioning for in-memory data analytics frameworks," in *IEEE International Conference on CLUSTER Computing*, 2016, pp. 110–119.

[24] K. M. Lee and K. M. Lee, "A locality sensitive hashing technique for categorical data," *Applied Mechanics and Materials*, vol. 244, pp. 3159–3164, 2013.

[25] J. Leskovec, A. Rajaraman, and J. D. Ullman, "Mining of massive datasets," 2012.

[26] 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*, 2012, pp. 2174–2178.

[27] X. He, J. Feng, B. Konte, S. T. Mai, and C. Plant, "Relevant overlapping subspace clusters on categorical data," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2014, pp. 213–222.

[28] T. Xiong, S. Wang, A. Mayers, and E. Monga, "Dhcc: Divisive hierarchical clustering of categorical data," *Data Mining and Knowledge Discovery*, vol. 24, no. 1, pp. 103–135, 2012.

[29] E. Cesario, G. Manco, and R. Ortale, "Top-down parameter-free clustering of high-dimensional categorical data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 12, pp. 1607–1624, 2007.

[30] O. Yim and K. T. Ramdeen, "Hierarchical cluster analysis: Comparison of three linkage measures and application to psychological data," *Tutorials in Quantitative Methods for Psychology*, vol. 11, no. 1, pp. 8–21, 2015.

[31] S. Niwattanakul, J. Singthongchai, E. Naenudorn, and S. Wanapu, "Using of jaccard coefficient for keywords similarity," *Lecture Notes in Engineering Computer Science*, vol. 2202, no. 1, 2013.

[32] P. Andritsos, P. Tsaparas, R. J. Miller, and K. C. Sevcik, "Limbo: Scalable clustering of categorical data," in *Advances in Database Technology - EDBT 2004, International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004, Proceedings*, 2004, pp. 123–146.

[33] J. Zhang, X. Yu, Y. Li, S. Zhang, Y. Xun, and X. Qin, "A relevant subspace based contextual outlier mining algorithm," *Knowledge-Based Systems*, vol. 99, no. C, pp. 1–9, 2016.

[34] Y. Yang, T. Rutayisire, C. Lin, T. Li, and F. Teng, "An improved cop-kmeans clustering for solving constraint violation based on mapreduce framework," *Fundamenta Informaticae*, vol. 126, no. 4, pp. 301–318, 2013.

[35] X. Li, *Parallel Algorithms for Hierarchical Clustering and Cluster Validity*. IEEE Computer Society, 1990.

[36] C. F. Olson, "Parallel algorithms for hierarchical clustering," *Pattern Analysis and Machine Intelligence IEEE Transactions on*, vol. 12, no. 11, pp. 1088–1092, 1995.

[37] Z. Du and F. Lin, "A novel parallelization approach for hierarchical clustering," *Parallel Computing*, vol. 31, no. 5, pp. 523–527, 2005.

[38] C. Jin, Z. Chen, W. Hendrix, A. Agrawal, and A. Choudhary, "Incremental, distributed single-linkage hierarchical clustering algorithm using mapreduce," in *Symposium on High PERFORMANCE Computing*, 2015, pp. 83–92.

[39] Y. Xun, J. Zhang, X. Qin, and X. Zhao, "Fidoop-dp: Data partitioning in frequent itemset mining on hadoop clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 101–114, 2017.

[40] S. Aridhi, L. D'Orazio, M. Maddouri, and E. Mephu, "A novel mapreduce-based approach for distributed frequent subgraph mining," *Actes De La Confrence Rfia*, 2014.

[41] X. Zhao, J. Zhang, and X. Qin, "knn-dp: Handling data skewness in knn joins using mapreduce," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 600–613, 2018.

[42] C. Jin, R. Liu, Z. Chen, W. Hendrix, A. Agrawal, and A. Choudhary, "A scalable hierarchical clustering algorithm using spark," in *IEEE First International Conference on Big Data Computing Service and Applications*, 2015, pp. 418–426.

[43] A. Krishnamurthy, S. Balakrishnan, M. Xu, and A. Singh, "Efficient active algorithms for hierarchical clustering," *Computer Science*, 2012.

Ning Pang received the MS in Computer Science and Technology in 2007 from Shanxi University, China. She is currently a Ph.D. student at Taiyuan University of Science and Technology(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, 2005,respectively. He is currently a Professor in the School of Computer Science and Technology at TYUST. His research interests include data mining, parallel computing, and celestial spectrum data analysis.

Chaowei Zhang received the BS in Software Engineering in 2014 from North University of China, China. He is currently a Ph.D. student in the Department of Computer Science and Software Engineering, Auburn University. His research interests include natural language processing, data mining and parallel computing.

Xiao Qin received the BS and MS degrees in Computer Science from Huazhong University of Science and Technology, China, and the Ph.D. degree in Computer Science from the University of Nebraska-Lincoln in 1992, 1999, and 2004, respectively. Currently, 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 CAREER Award in 2009. He is a senior member of the IEEE.