Chapter 11

Big Data Clustering

Hanghang Tong
IBM T. J. Watson Research Center
Yorktown Heights, NY
htong@us.ibm.com

U Kang
KAIST
Seoul, Korea
ukang@cs.cmu.edu

11.1 Introduction 259
11.2 One-Pass Clustering Algorithms 260
11.2.1 CLARANS: Fighting with Exponential Search Space 260
11.2.2 BIRCH: Fighting with Limited Memory 261
11.2.3 CURE: Fighting with the Irregular Clusters 263
11.3 Randomized Techniques for Clustering Algorithms 263
11.3.1 Locality-Preserving Projection 264
11.3.2 Global Projection 266
11.4 Parallel and Distributed Clustering Algorithms 268
11.4.1 General Framework 268
11.4.2 DBDC: Density-Based Clustering 269
11.4.3 ParMETIS: Graph Partitioning 269
11.4.4 PKMeans: K-Means with MapReduce 270
11.4.5 DisCo: Co-Clustering with MapReduce 271
11.4.6 BoW: Subspace Clustering with MapReduce 272
11.5 Conclusion 274
Bibliography 274

11.1 Introduction

With the advance of Web2.0, the data size is increasing explosively. For example, Twitter data spans several terabytes; Wikipedia data (e.g., articles and authors) is of similar size; web click-through data is reported to reach petabyte scale [36]; Yahoo! web graph in 2002 has more than 1 *billion* nodes and almost *7 billion* edges [27].

On the other hand, many data clustering algorithms have a high intrinsic time complexity. For example, the classic k-means clustering is NP-hard even when $k = 2$. The normalized cut (NCut), a representative spectral clustering algorithm, is also NP-hard [43]. Therefore, a key challenge for data clustering lies in its scalability, that is, how we can speed up/scale up the clustering algorithms with the minimum sacrifice to the clustering quality.

At the high level, many data clustering algorithms have the following procedure: after some

initialization (e.g., randomly choose the k cluster centers in k-means), it takes some iterative process until some convergence criteria is met. In each iteration, it adjusts/updates the cluster membership for each data point (e.g., assign, it to the closest cluster centers in k-means). Therefore, in order to speed up/scaleup such clustering algorithms, we have three basic ways: (1) by reducing the iteration number (e.g., one-pass algorithm) [47, 22], (2) by reducing the access to the data points (e.g., by randomized techniques) [25, 38], and (3) by distributing/parallelizing the computation [28, 13]. In this chapter, we will review some representative algorithms for each of these categories.

Notice that another important aspect related to the scalability is for the stream data, that is, how we can adopt the batch-mode data clustering algorithms in the case the data arrives in the stream mode, so that ideally the algorithms scale well with respect to the size of the new arrival data as opposed to that of the entire data set [16, 4, 35]. These issues will be discussed independently in the chapter on density-based clustering.

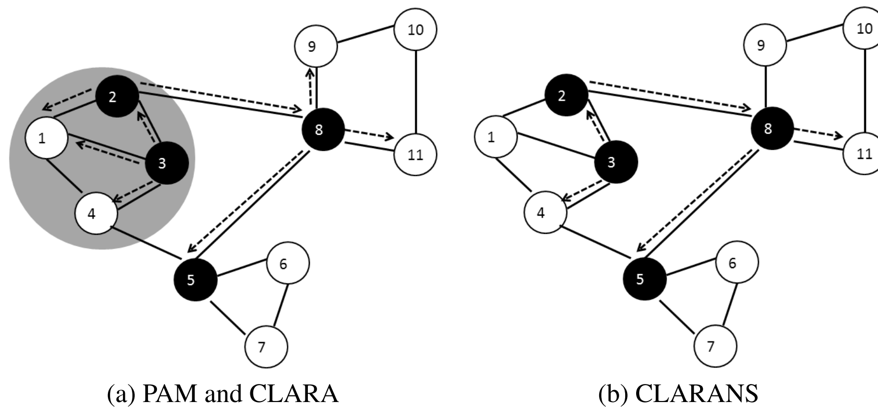
11.2 One-Pass Clustering Algorithms

In this section, we will review some earlier, classic clustering algorithms which are designed for large-scale data sets. CLARANS [39, 40] is one of the earliest algorithms to use randomized search to fight with the exponential search space in the K-medoid clustering problem. In BIRCH [47], a new data structure called clustering feature tree (CF-tree) was proposed to reduce the I/O cost when the input data set exceeds the memory size. Finally, CURE [22] uses multiple representative data points for each cluster in order to capture the irregular clustering shapes and it further adopts sampling to speed up the hierarchical clustering procedure. Thanks to these techniques, we can often largely reduce the iteration number in the clustering procedure. In some extreme cases, they often generate pretty good clustering results by one pass over the input data set. We collectively call these algorithms as one-pass clustering algorithms.

11.2.1 CLARANS: Fighting with Exponential Search Space

Let us start with one of the earliest representative clustering algorithms, i.e., Partitioning Around Medoids (PAM) [32]. The basic idea of PAM is to choose one representative point from each cluster (e.g., the most central data point in the cluster), which is referred to as a *medoid*. If we want to find k clusters for n data points, the algorithm essentially tries to find the k best medoids. Once such optimal k-medoid is found, the remaining (nonmedoid) data points are assigned to one of the selected medoids according to their distance to the k-medoid. To this end, it follows an iterative procedure where in each iteration, it tries to replace one of the current medoids by one of the nonmedoid data points. It is easy to see that the computational cost for PAM is high: first of all, in each iteration, we need to check all the $(n-k)\times k$ possible pairs; second, the overall search space is exponential, i.e., we have $\binom{n}{k}$ possible k-medoid assignment. To address this issue, Clustering Large Applications (CLARA) [32] relies on *sampling* to reduce the search space. Instead of trying to find the optimal k-medoid from the original *entire* data set as in PAM, CLARA first takes a sample of k from the original n data points and then calls PAM on these $O(k)$ sampled data points to find k-medoid. Once the optimal k-medoid is found, the remaining (nonsampled) data points are assigned to one of these k clusters based on their distance to the k-medoid.

Conceptually, both PAM and CLARA can be regarded as graph-searching problems. In this graph, each node is a possible clustering solution (i.e., a k-medoid), and the two nodes are linked to each other if they only differ in 1-out-of-k medoids. PAM starts with one of the randomly chosen



(a) PAM and CLARA (b) CLARANS

FIGURE 11.1: Each node in the graph is a possible clustering assignment (i.e., k-medoid); and each link means that two nodes differ in only one medoid. PAM (a) starts from a randomly selected node (e.g., node 3, dark), and greedily moves to the next best neighbor (e.g., node 2). In each iteration, it searches all the possible neighbors (dashed arrows). It tries to search over the entire graph. CLARA first samples a subgraph (shadowed area) and restricts the search procedure within this subgraph. CLARANS also searches the entire graph as in PAM. But in each iteration, it only searches only randomly sampled neighbors of the current node (dashed arrow).

nodes in the conceptual graph, and it greedily moves to one of its neighbors until it cannot find a better neighbor. CLARA aims to reduce the search space by only searching a subgraph that is induced by the sampled $O(k)$ data points.

Based on this observation, Clustering Large Applications based on Randomized Sampling (CLARANS) [39, 40] has been proposed to further improve the efficiency. As in PAM, CLARANS aims to find a local optimal solution by searching the entire graph. But unlike in PAM, in each iteration, it checks only a sample of the neighbors of the current node in the graph. Notice that both CLARA and CLARANS use sampling techniques to reduce the search space. But they conduct the sampling in the different ways. In CLARA, the sampling is done at the beginning stage to restrict the entire search process within a particular subgraph; whereas in CLARANS, the sampling is conducted *dynamically* at each iteration of the search process. The empirical evaluation in [39] shows that such dynamic sampling strategy in CLARANS leads to further efficiency improvement over CLARA. Figure 11.1 provides a pictorial comparison between PAM, CLARA and CLARANS.

11.2.2 BIRCH: Fighting with Limited Memory

When the data size exceeds the available memory amount, the I/O cost may dominate the in-memory computational time, where CLARANS and its earlier versions (e.g., PAM, CLARA) suffer. Balanced Iterative Reducing and Clustering using Hierarchies (BIRCH) [47] was one of the first methods to address this issue, and explicitly tries to reduce the I/O costs given the limited amount of memory.

The key of BIRCH is to introduce a new data structure called *clustering feature* (CF) as well as CF-tree. Therefore, before we present the BIRCH algorithm, let us take a short detour to introduce these concepts.

CF can be regarded as a concise summary of each cluster. This is motivated by the fact that not every data point is equally important for clustering and we cannot afford to keep every data point in the main memory given that the overall memory is limited. On the other hand, for the purpose of

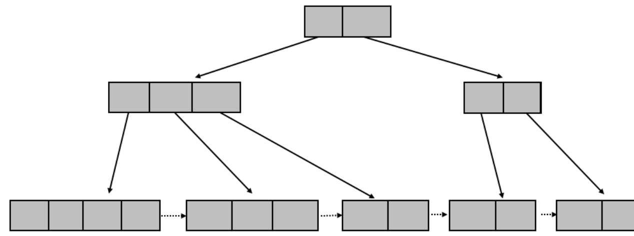


FIGURE 11.2: An illustration of the CF-tree. Each node represents a cluster which consists of up to B subclusters. Due to the additive property of CF, the CF of the parent node is simply the sum of the CFs of its children. All the leaf nodes are chained together for efficient scanning. The diameter of the subcluster in the leaf node is bounded by the threshold T. The bigger T is, the smaller the tree is.

clustering, it is often enough to keep up to the second order of data moment. In order words, CF is not only efficient, but also sufficient to cluster the entire data set.

To be specific, CF is a triple $< N, LS, SS >$ which contains the number of the data points in the cluster (i.e., the zero-order moment N), the linear sum of the data points in the cluster (i.e., the first-order moment LS), and the square sum of the data points in the cluster (i.e., the second order moment SS). It is easy to check that CF satisfies the additive property; that is, if we want to merge two existing clusters, the CF for the merged cluster is simply the sum of the CFs of the two original clusters. This feature is critical as it allows us to easily merge two existing clusters without accessing the original data set.

CF-tree is a height-balanced tree which keeps track of the hierarchical clustering structure for the entire data set. Each node in the CF-tree represents a cluster which is in turn made up of at most B subclusters, where B is the so-called balancing factor. All the leaf nodes are chained together for the purpose of efficient scanning. In addition, for each subcluster in the leaf node, its diameter is upper-bounded by a second parameter T (the threshold). Apparently, the larger T is, the smaller the CF-tree is in terms of the tree height. The insertion on CF-tree can be performed in a similar way as the insertion in the classic B-tree. Figure 11.2 presents a pictorial illustration of the CF-tree.

There are two main steps in the BIRCH algorithm. First, as it scans the input data points, it builds a CF-tree by inserting the data points with a default threshold $T = 0$. By setting the threshold $T = 0$, we treat each data point as an individual cluster at the leaf node. Thus, the size of the resulting CF-tree might be very large, which might lead to an out-of-memory issue in the middle of the scanning process. If such a situation happens, we will need to increase the threshold T and rebuild the CF-tree by reinserting all the leaf nodes from the old CF-tree. By grouping close data points together, the resulting CF-tree builds an initial summary of the input data set which reflects its rough clustering structure.

Due to the skewed input order and/or the splitting effect by the page size, the clustering structure from the initial CF-tree might not be accurate enough to reflect the real underlying clustering structure. To address this issue, the second key step ("global clustering") tries to cluster all the subclusters in the leaf nodes. This is done by converting a subcluster with n' data points n' times at the centroid and then running either an agglomerative hierarchical clustering algorithm or a modified clustering algorithm.

Between these two main steps, there is an optional step to refine the initial CF-tree by re-inserting its leaf entries ("tree condensing"). Usually, this leads to a much more compact CF-tree. After the global clustering step, there is another optional step ("clustering refinement"), which re-assigns all the data points based on the cluster centroid produced by the global clustering step. Figure 11.3 summarizes the overall flowchart of the BIRCH algorithm.

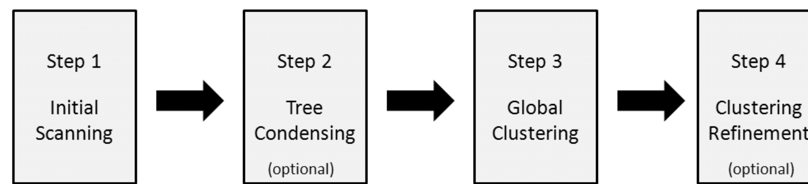


FIGURE 11.3: The flowchart of BIRCH algorithm.

The empirical evaluation in [47] indicates that BIRCH consistently outperforms the previous method CLARANS in terms of both time and space efficiency. It is also more robust to the ordering of the input data sequence. BIRCH offers some additional advantages, e.g., being robust to outliers and being more easily parallelized.

11.2.3 CURE: Fighting with the Irregular Clusters

In both CLARANS and BIRCH, we use one single data point to represent a cluster. Conceptually, we implicitly assume that each cluster has a spherical shape, which may not be the case in some real applications where the clusters can exhibit more complicated shapes. At the other extreme, we can keep all the data points within each cluster, whose computational as well as space cost might be too high for large data sets. To address this issue, clustering using representatives (CURE) [22] proposes to use a set of well-scattered data points to represent a cluster.

CURE is essentially a hierarchical clustering algorithm. It starts by treating each data point as a single cluster and then recursively merges two existing clusters into one until we have only k clusters. In order to decide which two clusters to be merged at each iteration, it computes the minimum distance between all the possible pairs of the representative points from the two clusters. CURE uses two major data structures to enable the efficient search. First, it uses a heap to track the distance of each existing cluster to its closest cluster. Additionally, it uses k-d tree to store all the representative points for each cluster.

In order to speed up the computation, CURE first draws a sample of the input data set and runs the above procedure on the sampled data. The authors further use Chernoff bound to analyze the necessary sample size. When the original data set is large, the different clusters might overlap each other, which in turn requires a large sample size. To alleviate this issue, CURE further uses partitions to speed up. To be specific, it first partitions the n' sampled data points into p partitions. Within each partition, it then runs a *partial* hierarchical clustering algorithm until either a predefined number of clusters is reached or the distance between the two clusters to be merged exceeds some threshold. After that, it runs another clustering pass on all the partial clusters from all the p partitions ("global clustering"). Finally, each nonsampled data point is assigned to a cluster based on its distance to the representative point ("labeling"). Figure 11.4 summarizes the flowchart of the CURE algorithm.

The empirical evaluation in [22] shows that CURE achieves lower execution time compared to BIRCH. In addition, as with BIRCH, CURE is also robust to outliers by shrinking the representative points to the centroid of the cluster with a constant factor.

11.3 Randomized Techniques for Clustering Algorithms

In the previous section, we have already seen how sampling can be used to speed up the clustering algorithms, e.g, to reduce the search space as in CLARA and CLARANS and to do pre-

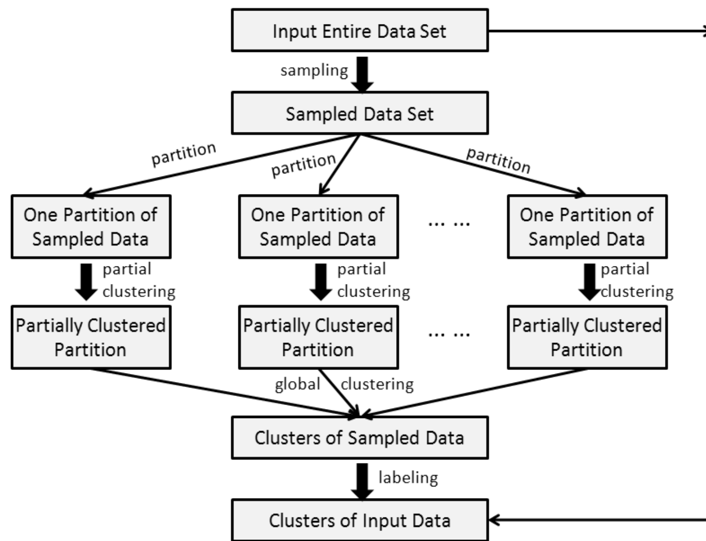


FIGURE 11.4: The flowchart of CURE algorithm.

clustering/partial clustering as in BIRCH and CURE. In this section, let us take one step further to see how randomized techniques can be used to address the scalability issue in clustering.

Suppose that we want to cluster n data points in d dimensional space into k clusters. We can represent these data points by an $n \times d$ data matrix A; and each row of this data matrix is a data point in d dimensional space. The basic idea of various randomized techniques is to reduce the scale of the input data matrix A, e.g., by transforming (embedding, projecting, etc.) it into a lower t-dimensional space ($t \ll d$) and then performing clustering on this reduced space. In this section, we will review two major types of such techniques.

11.3.1 Locality-Preserving Projection

In the locality-preserving projection (also referred to as random projection) methods, after we project the original d-dimensional data set into a lower dimensional space, we want the *pair-wise distance* to be preserved in a probabilistic sense. In many clustering algorithms, it is mainly the pair-wise distance measure that determines the clustering structure. Thus we would expect that the clustering results in the projected subspace would provide a reasonably good approximation to that in the original space.

In addition to clustering, random projection itself has a broad applicability in various data mining tasks (e.g., similarity search, near-duplicate detection). Here, we will first provide some general introduction to the typical random projection algorithms and then introduce how to apply them in the clustering setting.

A classic result for the locality-preserving projection comes from Johnson and Lindenstrauss [25], which can be formally stated as follows:

Lemma 11.3.1 *Assume* $\varepsilon > 0$*, and* n *is an integer. Let* t *be a positive integer such that* $t \geq t_0 = O(log(n)/\varepsilon^2)$*. For any set* P *with* n *data points in* R^d *space, there exist* $f : R^d \to R^t$ *such that for all* $u, v \in P$*, we have*

$$(1-\varepsilon)\|u-v\|^2 \leq \|f(u)-f(v)\|^2(1+\varepsilon)\|u-v\|^2$$

At the algorithmic level, such random projection (also referred to as "JL-embedding") can be performed by a linear transformation of the original data matrix A. Let R be a $d \times t$ rotation matrix with its elements $R(i, j)$ being independent random variables. Then $\tilde{A} = A \cdot R$ is the projected data matrix,[1] and each of its row is a vector in t dimensional space which is the projection of the corresponding data point in d dimensional space.

Different random projection algorithms differ in terms of different ways to construct such a rotation matrix R. Earlier methods suggest $R(i, j)$ as being independent Normal random variables with mean 0 and variance 1. In [1], the authors propose two database-friendly ways to construct the rotation matrix R. The first method sets $R(i, j) = \pm 1$ with equal probabilities of 0.5. Alternatively, we can also set $R(i, j) = \pm\sqrt{3}$ with equal probabilities of $1/6$ and $R(i, j) = 0$ with the probability of $2/3$.

Once we have projected the original data matrix A into a low-dimensional space, the clustering can be performed in the projected subspace $\tilde{A}$. In [18], the authors assume that the projected data points form a Gaussian-mixture model and further propose using the expectation-maximization (EM) algorithm to find the soft clusters; that is, each data point i is assigned a cluster membership probability $p(l|i, \theta)$, where l is the lth cluster and θ is a parameter for the underlying Gaussian-mixture model. The rationality of this approach can be traced back to [14], which found that by random projection, the original high-dimensional distribution tends to be more like Gaussian distribution in the projected subspace. It was also found in [10] that eccentric clusters can be reshaped to be more spherical by random projection.

Due to its random nature, the clustering result generated by the above procedure (RP+EM) might be highly unstable; that is, different runs of random projections might lead to very different clustering assignment despite the fact that each of them might *partially* reveal the true clustering structure. Based on this observation, the authors in [18] further propose running this RP+EM procedure multiple times followed up by an ensemble step. To be specific, after we run RP+EM multiple times (say T in total), we construct an $n \times n$ similarity/affinity matrix P, where each element indicates the average probability that the corresponding two data points are assigned to the same cluster, i.e., $P(i, j) = \frac{1}{T}\sum_{t=1}^{T}\sum_{l=1}^{k} p(l|i, \theta_t) \times p(l|j, \theta_t)$, where l and t are the indices for clusters and different runs, respectively. Then, an agglomerative clustering algorithm is run based on this affinity matrix P to generate the final k clusters. The empirical evaluations show that such ensemble framework leads to much more robust clustering results. Figure 11.5 summarizes the overall flowchart of this clustering ensemble framework based on RP+EM.

Given that the k-means clustering is NP-hard even if $k = 2$, many recent research works have focused on the so-called $\gamma-$approximation algorithms. Let us introduce an $n \times k$ cluster indictor matrix X, where each row of X has only one nonzero element; and $X(i, j) \neq 0$ indicates that the data point i belongs to the jth cluster.[2] Optimal k-means searches for an indicator matrix X_{opt} such that $X_{opt} = argmin_{\mathcal{X}} \|A - XX^TA\|_{fro}^2$, where $\mathcal{X}$ is the set of all valid $n \times k$ indictor matrices.

For any $\gamma > 1$ and the failure probability $0 \leq \delta_\gamma < 1$, a $\gamma-$approximation algorithm finds an indicator matrix X_γ such that with the probability at least $1 - \delta_\gamma$, we have $\|A - XX^TA\|_{fro}^2 \leq \gamma \cdot min_{\mathcal{X}} \|A - XX^TA\|_{fro}^2$. In [33], the authors propose the first *linear* time $\gamma-$approximation algorithm by sampling. More recently, the authors in [3] propose further speeding up the computation by running such a $\gamma-$approximation algorithm on the projected subspace.

[1] Depending on the way we construct the rotation matrix, we might need to do a constant scaling on the elements of the projected matrix $\tilde{A}$.

[2] The actual value of such non-zero elements is determined by the size of the clusters. If we have n_j data points in the jth cluster, we have $X(i, j) = 1/\sqrt{n_j}$.

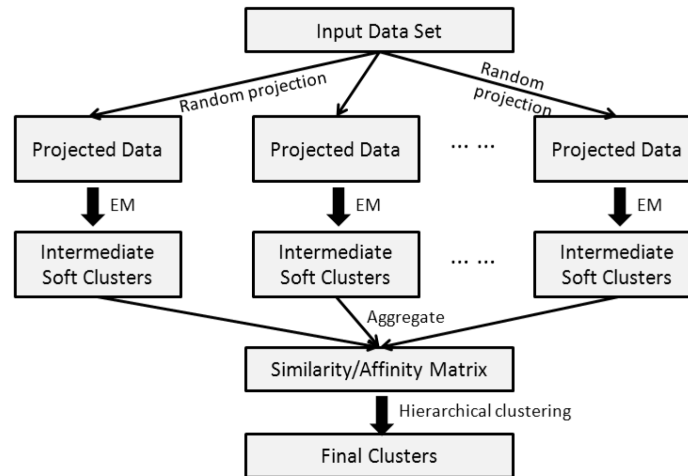


FIGURE 11.5: The flowchart of clustering ensemble based on Random Projection and EM algorithm.

11.3.2 Global Projection

In the locality-preserving projection, for any *two* data points, we want their pair-wise distance approximately unchanged before and after the projection. In global projection, for *each* data point, we want its projection to be as close as possible to the original data point. If we use the Euclidian distance, it is equivalent to saying that we want to minimize the Frobenius norm of $(\tilde{A} - A)$, where $\tilde{A}$ is the approximation of the original data matrix A.

We can represent a global projection using the notation of matrix low-rank approximation. Formally, a rank-c approximation of matrix A is a matrix $\tilde{A}$; $\tilde{A} = L \cdot M \cdot R$ where L, M, and R are of rank-c; and $\|\tilde{A} - A\|$ is small. Such a low-rank approximation often provides a powerful tool for clustering. For example, for the spatial data, the left matrix L often provides a good indicator for the cluster-membership; and the row of the right matrix R provides the description of the corresponding cluster. For the bipartite graph data, we can represent it by its adjacency matrix A; then the left and right matrices are often good indicators for row and column cluster memberships; and the middle matrix M indicates the interaction between the row and the column clusters. We refer the readers to Chapter 7 for more details on the clustering algorithms based on matrix low-rank approximation. Here, we introduce how to get such low-rank approximation efficiently so that the overall clustering procedure is efficient and scalable.

Depending on the properties of those matrices (L, M, and R), many different approximations have been proposed in the literature. For example, in singular value decomposition (SVD) [21], L and R are orthogonal matrices whose columns/rows are singular vectors and M is a diagonal matrix whose diagonal entries are singular values. Among all the possible rank-c approximations, SVD gives the best approximation in terms of squared error. However, SVD is usually dense, even if the original matrix is sparse. Furthermore, the singular vectors are abstract notions of best orthonormal basis, which is not intuitive for the interpretation.

To address the computational challenges in SVD, sparsification was proposed in [1]. The basic idea is to randomly set a significant portion of the entries in A as zeros and rescale the remaining entries; and to run SVD on the resulting sparse matrix instead. For instance, with uniform sampling, each entry $A(i,j)$ is set as zero with the probability of $1-1/s (s > 1)$ and is scaled as $sA(i,j)$ with the probability of $1/s$.[3] The resulting sparse matrix $\bar{A}$ can be viewed as a perturbated version of the

[3]The authors in [1] also proposed a non-uniform sampling procedure.

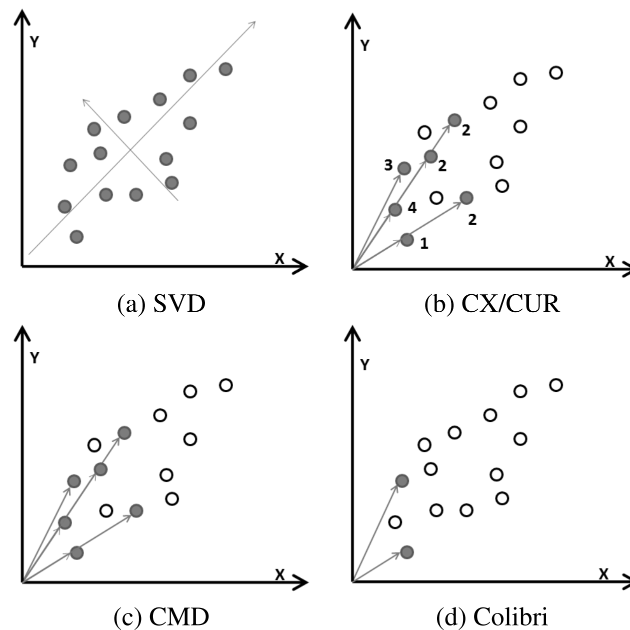


FIGURE 11.6: An illustrative comparison of SVD, CX/CUR, CMD, and Colibri. Each dot is a data point in 2-D space. SVD (a) uses all the available data points to generate optimal projection directions. CX/CUR (b) uses actual sampled data points (dark dots) as projection directions and there are a lot of duplicates. CMD (c) removes duplicate sampled data points. Colibri (d) further removes all the linearly correlated data points from sampling.

original matrix A by adding a random matrix E to it. From the random matrix theory [20], it is well known that the norm of a random matrix is well-bounded. As a result, the SVD on this sparse matrix $\bar{A}$ provides a *near-optimal* low-rank approximation of the original data matrix A, at the benefit of significantly speeding up the computation.

Notice that the sparse SVD provides computational gain compared with the original exact SVD, but it does not address the issue of the space cost or the interpretability of the resulting low-rank approximation. For example, as shown in [44], these methods often need a huge amount of space. More recently, the so-called *example-based low-rank approximations* have started to appear, such as CX/CUR [15], CMD [44] and Colibri [45], which use the actual columns and rows of the data matrix A to form L and R. The benefit is that they provide an intuitive as well as a sparse representation, since L and R are directly sampled from the original data matrix. However, the approximation is often sub-optimal compared to SVD and the matrix M is no longer diagonal, which means a more complicated interaction.

In CX decomposition, we first randomly sample a set of columns C from the original data matrix A, and then project the A into the column space of C, i.e., $\tilde{A} = CC^{\dagger}A$, where $C^{\dagger}$ is the Moore-Penrose pseudo inverse of C. In CUR decomposition, we also do sampling on the row side to further reduce the space cost. One drawback of the original CX/CUR is the repeated sampling, i.e., some sampled columns/rows are duplicate. Based on this observation, CMD tries to remove such duplicate rows and columns before performing the projection. In [45], the authors further propose removing all the linearly correlated sampled columns. Since it does not affect the column/row space if we remove the linearly correlated columns/rows, these methods (CMD and Colibri) lead to the same approximation accuracy as CX/CUR, but require much less space and cost. Figure 11.6 provides an illustrative comparison of these different projection methods.

11.4 Parallel and Distributed Clustering Algorithms

In this section, we survey parallel and distributed clustering algorithms to handle big data. In contrast to the typical single machine clustering, parallel and distributed algorithms use multiple machines to speed up the computation and increase the scalability. The parallel and distributed algorithms are divided into two groups: traditional memory-based algorithms [13, 19, 5, 6, 41] and modern disk-based algorithms [11, 9, 29, 26].

The traditional memory-based algorithms assume that the data fit in the memory of multiple machines. The data is distributed over multiple machines, and each machine loads the data into memory.

Many of the modern disk-based algorithms use MapReduce [11], or its open-source counterpart Hadoop, to process disk resident data. MapReduce is a programming framework for processing huge amounts of data in a massively parallel way. MapReduce has two major advantages: (a) the programmer is oblivious of the details of the data distribution, replication, load balancing, etc., and (b) the programming concept is familiar, i.e., the concept of functional programming. Briefly, the programmer needs to provide only two functions, a *map* and a *reduce*. The typical framework is as follows [34]: (a) the *map* stage sequentially passes over the input file and outputs (key, value) pairs; (b) the *shuffling* stage groups all values by key, and (c) the *reduce* stage processes the values with the same key and outputs the final result. Optionally, *combiners* can be used to aggregate the outputs from local mappers. MapReduce has been used widely for large scale data mining [29, 26].

We describe both the traditional and modern parallel and distributed clustering algorithms. We first describe the general principle of parallel and distributed algorithms. Then we survey traditional algorithms including DBDC and ParMETIS . Finally we survey modern, disk-based MapReduce algorithms including PKMeans, DisCo, and BoW.

11.4.1 General Framework

Most parallel and distributed clustering algorithms follow the general framework depicted in Figure 11.7.

1. **Partition.** Data are partitioned and distributed over machines.
2. **Local Clustering.** Each machine performs local clustering on its partition of the data.
3. **Global Clustering.** The cluster information from the previous step is aggregated globally to produce global clusters.
4. **Refinement of Local Clusters.** Optionally, the global clusters are sent back to each machine to refine the local clusters.

Some algorithms (e.g. PKMeans in Section 11.4.4 and DisCo in Section 11.4.5) iteratively perform steps 3 and 4 until the quality of clusters becomes reasonably good. One of the main challenges of a parallel and distributed clustering algorithm is to minimize data traffic between the steps 2 and 3. Minimizing the traffic is important since the advantage of a parallel and distributed clustering algorithms comes from the fact that the output from the local clustering step is much smaller than the raw data. We will see how different algorithms use different strategies to minimize the data traffic.

Another issue in parallel and distributed clustering is the lower accuracy compared to the serial counterpart. There are two main reasons for the lower accuracy. First, each machine performing the local clustering might use a different clustering algorithm than the serial counterpart due to heavy communication costs. Second, even though the same algorithm is used for the local clustering step

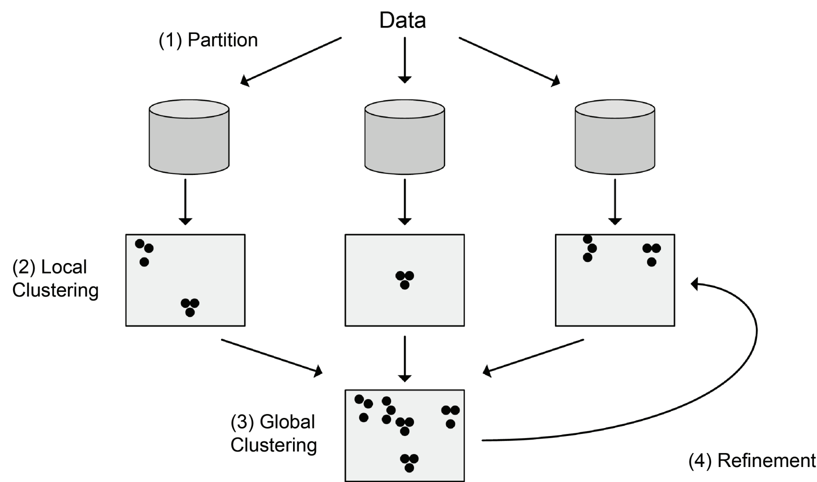


FIGURE 11.7: The general framework of most parallel and distributed clustering algorithms. (1) Data is partitioned. (2) Data is locally clustered. (3) Local clusters are merged to make global clusters. (4) Optionally, local clusters are refined using the global clusters.

as well as the serial algorithm, the divided data might change the final aggregated clusters. We will survey the clustering quality, too, in the following.

11.4.2 DBDC: Density-Based Clustering

DBDC [24] is a density-based distributed clustering algorithm. Density-based clustering aims to discover clusters of arbitrary shape. Each cluster has a density of points which is considerably higher than outside of the cluster. Also, the density within the areas of noise is lower than the density in any of the clusters.

DBDC is an exemplary algorithm that follows the general framework given in Section 11.4.1. Initially the data is partitioned over machines. At the local clustering step, each machine performs a carefully designed clustering algorithm to output a set of a small number of representatives which has an accurate description of local clusters. The representatives are merged in the global clustering step using DBSCAN [17], a single-machine density-based clustering algorithm. Then the global clusters are sent back to all clients sites which relabel all objects located on their site independently of each other.

The experimental results clearly show the advantage of the distributed clustering. The running time of DBDC is more than 30 times faster than the serial clustering counterpart. Moreover, DBDC yields almost the same clustering quality as the serial algorithm.

11.4.3 ParMETIS: Graph Partitioning

ParMETIS [31] is a parallel graph partitioning algorithm. Given a vertex and edge weighted graph, k-way graph partitioning algorithm partitions the vertices into k subsets so that the sum of the weight of the vertices in each subset is roughly the same, and the weight of the edges whose incident edges belong to different subsets is small. For example, Figure 11.8 shows an example of 3-way graph partitioning of a graph with 14 vertices. The graph partitioning is essentially a clustering problem where the goal is to find good clusters of vertices.

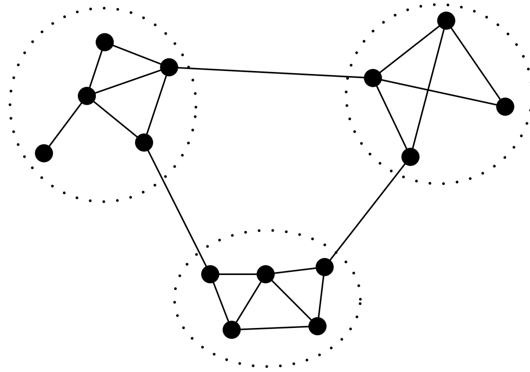


FIGURE 11.8: The *k*-way graph partitioning problem partitions the vertices into *k* subsets so that the sum of the weight of the vertices in each subset is roughly the same, and the weight of the edges whose incident edges belong to different subsets is small. The figure shows 3-way partitioning of a graph with 14 vertices. A dotted circle represents a partition.

There have been many works on the graph partitioning. One of the state-of-the-art graph partitioning algorithms is METIS [30]. ParMETIS is a parallel algorithm of METIS for distributed memory system.

METIS is a multilevel partitioning algorithm. There are three main phases in the METIS. In the first *coarsening* phase, maximal matching on the original graph is performed. Then, the matched vertices are collapsed together to create a smaller graph. This process is repeated until the number of vertices is small enough. In the second *partitioning* phase, *k*-way partitioning of the coarsened graph is performed by a multilevel recursive bisection algorithm. In the third *uncoarsening* phase, the partitioning from the second phase is projected back to the original graph by a greedy refinement algorithm.

ParMETIS performs the three phases of METIS using memory-based distributed systems. ParMETIS does not follow the general framework in Section 11.4.1, since the clustering is mainly based on the coarsening and the uncoarsening operations which are graph operations different from clustering operations. Initially each processor receives an equal number of vertices. At the coarsening phase, ParMETIS first computes a coloring of a graph and then computes a global graph incrementally matching only vertices of the same color one at a time. At the partitioning phase, the coarsened graph is broadcasted to all the machines. Each machine performs recursive bisection by exploring only a single path of the recursive bisection tree. At the uncoarsening phase, vertices of the same color are considered for a movement; subsets of the vertices that lead to a reduction in the edge-cut are moved. When moving vertices, the vertices themselves do not move until the final step: only the partition number associated with each vertex moves to minimize communication cost.

Experiments were performed on a 128-processor Cray T3D parallel computer with distributed memories. In terms of partition quality, the edge-cut produced by the parallel algorithm is quite close to that produced by the serial algorithm. The running time of the ParMETIS using 128 processors was from 14 to 35 times faster than the serial algorithm.

11.4.4 PKMeans: *K*-Means with MapReduce

MapReduce has been used for many clustering algorithms. Here we describe PKMeans [48], a *k*-means [37, 2, 23, 46] clustering algorithm on MapReduce. Given a set of *n* objects and the number of cluster *k*, the *k*-means algorithm partitions the objects so that the objects within a cluster are more

similar to each other than the objects in different clusters. Initially, k objects are randomly selected as the center of clusters. Then the k-means algorithm performs the following two tasks iteratively:

- **Step 1.** Assign each object to the cluster whose center is the closest to the object.
- **Step 2.** For each cluster, update the center with the mean of the objects in the cluster.

PKMeans uses MapReduce to distribute the computation of k-means. PKMeans follows the general framework shown in Section 11.4.1. The partitioning is implicitly performed when the data is uploaded to the distributed file system (e.g., GFS or HDFS) of MapReduce. The local clustering, which corresponds to Step 1 above, is performed in the mapper. The global clustering, which corresponds to Step 2 above, is performed in the combiner and the reducer.

Specifically, the objects data $x_1...x_n$, for which we assume d-dimensional points, are distributed in the HDFS. The centers $y_1...y_k$, which are also d-dimensional points, of clusters are given to the mapper in the format of $\{i, y_i\}$ by the parameter passing mechanism of Hadoop. The details of the mapper, the combiner, and the reducer are as follows:

- Mapper: Read the object data x_i, and find the center y_j which is closest to x_i. That is, $j = \text{argmin}_l ||x_i - y_l||_2$. Emit $< j, x_i >$.
- Combiner: Take $< j, \{x_i\} >$ and emit $< j, \sum x_i, num >$ where *num* is the number of objects that the combiner received with the key j.
- Reducer: Take $< j, \{(\sum x_i, num)\} >$, and compute the new center. Emit the new center $< j, y_j >$ of the cluster.

The outputs of the reducer are the centers of clusters which are fed into the mapper of the next iteration. We note that the combiner greatly decreases the amount of intermediate data by emitting only the sum of the input data.

Experimental results show that PKMeans provides good *speed up*, *scale up*, and *size up*. The speed up is evaluated by the ratio of running time while keeping the dataset constant and increasing the number of machines in the system. PKMeans shows near-linear speed up. The scale up measures whether x-times larger system can perform x-times larger job in the same run-time as the original system. PKMeans shows a good scale up, better than 0.75 for 4 machines. In addition, PKMeans has a linear size up: given the fixed number of machines, the running time grows linearly with the data size. Finally, PKMeans is an exact algorithm, and thus the quality of the clustering is the same as that of the serial k-means.

11.4.5 DisCo: Co-Clustering with MapReduce

DisCo [42] is a distributed co-clustering algorithm with MapReduce. Given a data matrix, co-clustering groups the rows and columns so that the resulting permuted matrix has concentrated nonzero elements. For example, co-clustering on a documents-to-words matrix finds document groups as well as word groups. For an $m \times n$ input matrix, co-clustering outputs the row and column labeling vector $\mathbf{r} \in \{1, 2, ..., k\}^m$ and $\mathbf{c} \in \{1, 2, ..., l\}^n$, respectively, and $k \times l$ group matrix G where k and l are the number of desired row and column partitions, respectively. Searching for an optimal cluster is NP-hard [12], and thus co-clustering algorithms perform a local search. In the local search, each row is iterated to be assigned to the best group that gives the minimum cost while the column group assignments are fixed. Then in the same fashion each column is iterated while the row group assignments are fixed. This process continues until the cost function stops decreasing.

There are two important observations that affect the design of the distributed co-clustering algorithm on MapReduce:

- The numbers k and l are typically small, and thus the $k \times l$ matrix G is small. Also, the row and the column labeling vectors r and c can fit in the memory.
- For each row (or column), finding the best group assignment requires only r, c, and G. It does not require other rows.

Exploiting the above two characteristics, G, r, and c are broadcast to mappers via parameter passing mechanism of MapReduce. Then each row (or column) can be independently processed to be assigned to the best group that minimizes the cost. Each row (or column) iteration requires a map and a reduce stage, and it follows the general framework shown in Section 11.4.1. The mapper reads each row and performs local clustering. The reducer performs global clustering by gathering local cluster information and outputs the updated G matrix and the r row-label vector (c column-label vector for the column iteration). DisCo minimizes network traffic by transferring only the label vectors and the G matrix between the mapper and the reducer; the matrix data are not transferred.

Experiments were performed on a 39-node Hadoop cluster on matrix data up to 2.5 million by 1.3 million. The performance, measured by the aggregated throughput, increased linearly with the number of machines. The quality of DisCo is the same as that of the serial algorithm since DisCo is an exact algorithm.

11.4.6 BoW: Subspace Clustering with MapReduce

BoW [7] is a distributed subspace clustering algorithm on MapReduce. BoW provides two subspace clustering algorithms on MapReduce: Parallel Clustering (ParC) and Sample-and-Ignore (SnI). ParC has three steps as illustrated in Figure 11.9: (1) partition the input data using mappers so that data with the same partition are aggregated to a reducer, (2) reducers find clusters in their assigned partitions using any subspace clustering algorithm (e.g., [8]) as a plug-in), and (3) merge the clusters from the previous step to get final clusters. Steps (1) and (2) are performed in a map and a reduce stage, respectively, and step (3) is done serially in a machine.

SnI uses sampling to minimize network traffic. SnI comprises two phases as depicted in Figure 11.10. In the first phase, (1) mappers randomly sample data (e.g., 10% of the data) and send

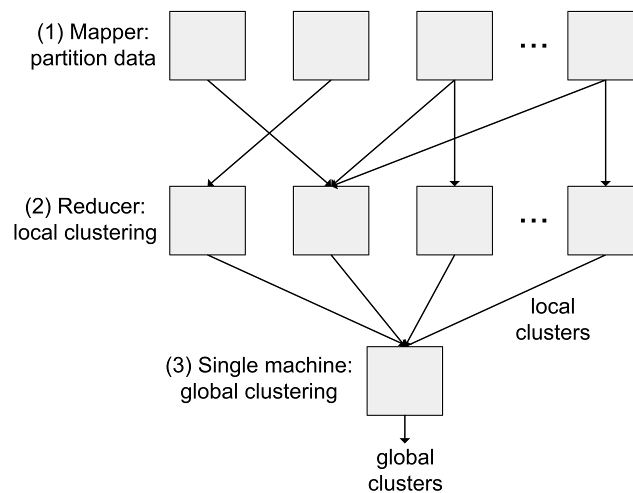


FIGURE 11.9: The ParC algorithm for subspace clustering. (1) Mappers partition the data. (2) Each reducer finds clusters. (3) A single machine collects and merges all the clusters from the output of the reducers to make final clusters.

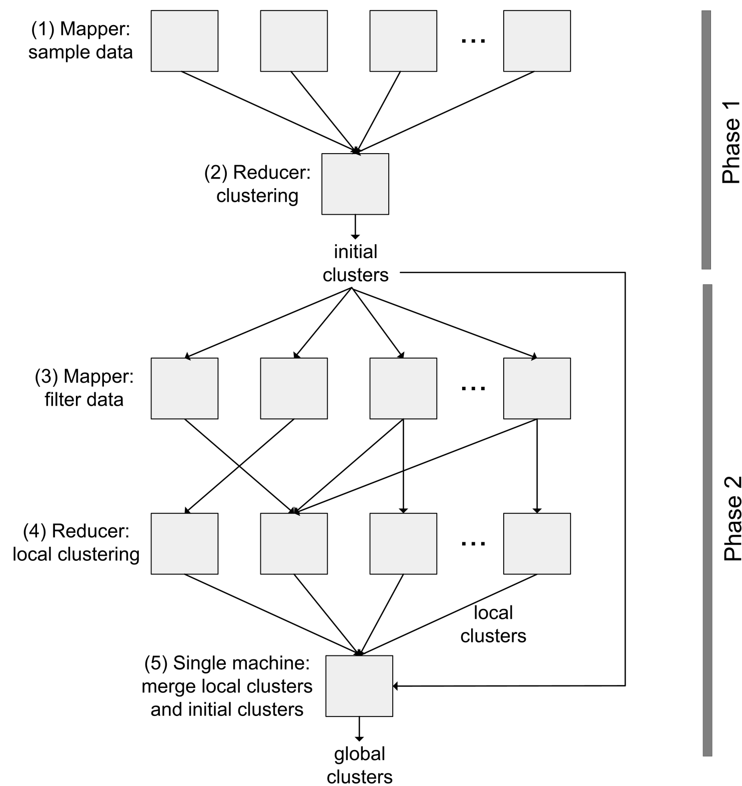


FIGURE 11.10: The SnI algorithm for subspace clustering. (1) Mappers sample the data. (2) A reducer finds initial clusters from the sampled data. (3) Mappers sends to reducers only the data which do not belong to the initial clusters from the second step. (4) Reducers find clusters. (5) A single machine combines the clusters from the second and the fourth steps to make final clusters.

results to a reducer. (2) The reducer runs a subspace clustering algorithm to find initial clusters. In the second phase, (3) mappers send to reducers only the data points which do not belong to the initial clusters found in the first phase, (4) the reducers find clusters, and (5) a single machine combines the clusters from the reducer with the initial clusters from the first phase to make final clusters.

ParC and the second phase of SnI follow the general framework described in Section 11.4.1: data is partitioned, data is locally clustered, local clusters are aggregated to make global clusters. The first phase of SnI is a preprocessing step to minimize network traffic.

ParC and SnI have their own pros and cons. In terms of the number of disk I/Os, ParC is better since it requires only one map-and-reduce step while SnI requires two map-and-reduce steps. However, in terms of the network traffic SnI is better due to the sampling in the first phase and the filtering in the second phase.

The main question is which algorithm runs faster? There are many parameters to consider to answer the question: e.g., number of reducers used, data size, disk speed, network speed, start-up cost, the plug-in algorithm cost, ratio of data transferred in the shuffling stage, and the sampling rate. BoW derives the cost (running time) as a function of the parameters to determine the algorithm that gives the minimum running time.

Experimental results show that BoW correctly chooses the best algorithm for different numbers of reducers. BoW also shows linear scalability on the data size. Finally, we note that both ParC and

SnI are not exact algorithms due to the clustering on the divided data. However, the quality of the clustering from either ParC or SnI is comparable to the serial version of the algorithm.

11.5 Conclusion

Given that (1) data size keeps growing explosively and (2) the intrinsic complexity of a clustering algorithm is often high (e.g., NP-hard), the scalability seems to be a "never-ending" challenge in clustering. In this chapter, we have briefly reviewed three basic techniques to speed up/scale up a data clustering algorithm, including (a) "one-pass" algorithms to reduce the iteration number in clustering procedure, (b) randomized techniques to reduce the complexity of the input data size, and (c) distributed and parallel algorithms to speed up/scale up the computation. A future trend is to integrate all these available techniques to achieve even better scalability.

Bibliography

[1] Dimitris Achlioptas and Frank McSherry. Fast computation of low-rank matrix approximations. *Journal of the ACM*, 54(2), 2007.

[2] Sanghamitra Bandyopadhyay, Chris Giannella, Ujjwal Maulik, Hillol Kargupta, Kun Liu, and Souptik Datta. Clustering distributed data streams in peer-to-peer environments. *Information Sciences*, 176(14):1952–1985, 2006.

[3] Christos Boutsidis, Anastasios Zouzias, and Petros Drineas. Random projections for k-means clustering. In *NIPS*, pages 298–306, 2010.

[4] Moses Charikar, Chandra Chekuri, Tomás Feder, and Rajeev Motwani. Incremental clustering and dynamic information retrieval. In *STOC*, pages 626–635, 1997.

[5] Wen-Yen Chen, Yangqiu Song, Hongjie Bai, Chih-Jen Lin, and Edward Y. Chang. Parallel spectral clustering in distributed systems. *IEEE Transactons on Pattern Analysis and Machine Intelligence*, 33(3):568–586, 2011.

[6] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-Reduce for machine learning on multicore. In *NIPS*, pages 281–288, 2006.

[7] Robson Leonardo Ferreira Cordeiro, Caetano Traina Jr., Agma Juci Machado Traina, Julio López, U. Kang, and Christos Faloutsos. Clustering very large multi-dimensional datasets with MapReduce. In *KDD*, pages 690–698, 2011.

[8] Robson Leonardo Ferreira Cordeiro, Agma J. M. Traina, Christos Faloutsos, and Caetano Traina Jr. Finding clusters in subspaces of very large, multi-dimensional datasets. In *ICDE*, pages 625–636, 2010.

[9] Abhinandan Das, Mayur Datar, Ashutosh Garg, and ShyamSundar Rajaram. Google news personalization: Scalable online collaborative filtering. In *WWW*, pages 271–280, 2007.

[10] Sanjoy Dasgupta. Experiments with random projection. In *UAI*, pages 143–151, 2000.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *OSDI*, pages 139–149, 2004.

[12] Inderjit S. Dhillon, Subramanyam Mallela, and Dharmendra S. Modha. Information-theoretic co-clustering. In *KDD*, pages 89–98, 2003.

[13] Inderjit S. Dhillon and Dharmendra S. Modha. A data-clustering algorithm on distributed memory multiprocessors. In *Large-Scale Parallel Data Mining*, pages 245–260, 1999.

[14] P. Diaconis and D. Freedman. Asymptotics of graphical projection pursuit. *Annals of Statistics*, 12(3):793–813, 1984.

[15] Petros Drineas, Ravi Kannan, and Michael W. Mahoney. Fast Monte Carlo algorithms for matrices III: Computing a compressed approximate matrix decomposition. *SIAM Journal of Computing*, 36(1):132–157, 2005.

[16] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Michael Wimmer, and Xiaowei Xu. Incremental clustering for mining in a data warehousing environment. In *VLDB*, pages 323–333, 1998.

[17] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231, 1996.

[18] Xiaoli Zhang Fern and Carla E. Brodley. Random projection for high dimensional data clustering: A cluster ensemble approach. In *ICML*, pages 186–193, 2003.

[19] George Forman and Bin Zhang. Distributed data clustering can be efficient and exact. *SIGKDD Explorations*, 2(2):34–38, 2000.

[20] Z. Fredi and J. Komlos. The eigenvalues of random symmetric matrices. *Combinatorica*, 1:233–241, 1981.

[21] G. H. Golub and C. F. Van-Loan. *Matrix Computations*, 2nd edition, The Johns Hopkins University Press, Baltimore, 1989.

[22] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. CURE: An efficient clustering algorithm for large databases. *Information Systems*, 26(1):35–58, 2001.

[23] Geetha Jagannathan and Rebecca N. Wright. Privacy-preserving distributed k-means clustering over arbitrarily partitioned data. In *KDD*, pages 593–599, 2005.

[24] Eshref Januzaj, Hans-Peter Kriegel, and Martin Pfeifle. DBDC: Density based distributed clustering. In *EDBT*, pages 88–105, 2004.

[25] W. B. Johnson and J. Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. *Contemporary Mathematics*, 26:189–206, 1984.

[26] U. Kang, Evangelos E. Papalexakis, Abhay Harpale, and Christos Faloutsos. Gigatensor: Scaling tensor analysis up by 100 times—Algorithms and discoveries. In *KDD*, pages 316–324, 2012.

[27] U. Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. GBASE: A scalable and general graph management system. In *KDD*, pages 1091–1099, 2011.

[28] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system. In *ICDM*, pages 229–238, 2009.

[29] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. Pegasus: Mining peta-scale graphs. *Knowledge and Information Systems*, 27(2):303–325, 2011.

[30] George Karypis and Vipin Kumar. Multilevel *k*-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

[31] George Karypis and Vipin Kumar. Parallel multilevel *k*-way partitioning for irregular graphs. *SIAM Review*, 41(2):278–300, 1999.

[32] Leonard Kaufman and Peter J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley & Sons, 1990.

[33] Amit Kumar, Yogish Sabharwal, and Sandeep Sen. A simple linear time (1+)-approximation algorithm for *k*-means clustering in any dimensions. In *FOCS*, pages 454–462, 2004.

[34] Ralf Lämmel. Google's MapReduce programming model—Revisited. *Science of Computer Programming*, 70:1–30, 2008.

[35] Jessica Lin, Michail Vlachos, Eamonn J. Keogh, and Dimitrios Gunopulos. Iterative incremental clustering of time series. In *EDBT*, pages 106–122, 2004.

[36] Chao Liu, Fan Guo, and Christos Faloutsos. BBM: Bayesian browsing model from petabyte-scale data. In *KDD*, pages 537–546, 2009.

[37] J.B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297. University of California Press.

[38] Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *KDD*, pages 169–178, 2000.

[39] Raymond T. Ng and Jiawei Han. CLARANS: A method for clustering objects for spatial data mining. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1003–1016, 2002.

[40] Raymond T. Ng and Jiawei Han. Efficient and effective clustering methods for spatial data mining. In *VLDB*, pages 144–155, 1994.

[41] Clark F. Olson. Parallel algorithms for hierarchical clustering. *Parallel Computing*, 21(8):1313–1325, 1995.

[42] Spiros Papadimitriou and Jimeng Sun. DisCo: Distributed co-clustering with Map-Reduce: A case study towards petabyte-scale end-to-end mining. In *ICDM*, pages 512–521, 2008.

[43] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.

[44] Jimeng Sun, Yinglian Xie, Hui Zhang, and Christos Faloutsos. Less is more: Compact matrix decomposition for large sparse graphs. In *SDM*, 2007.

[45] Hanghang Tong, Spiros Papadimitriou, Jimeng Sun, Philip S. Yu, and Christos Faloutsos. Colibri: Fast mining of large static and dynamic graphs. In *KDD*, pages 686–694, 2008.

[46] Jaideep Vaidya and Chris Clifton. Privacy-preserving *k*-means clustering over vertically partitioned data. In *KDD*, pages 206–215, 2003.

[47] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. BIRCH: An efficient data clustering method for very large databases. In *SIGMOD Conference*, pages 103–114, 1996.

[48] Weizhong Zhao, Huifang Ma, and Qing He. Parallel *k*-means clustering based on MapReduce. In *CloudCom*, pages 674–679, 2009.