Small Journal Name, , 1–40 ()

BIRCH: A New Data Clustering Algorithm and Its Applications

TIAN ZHANG, RAGHU RAMAKRISHNAN, MIRON LIVNY zhang,raghu,miron@cs.wisc.edu

Computer Sciences Department, University of Wisconsin, Madison, WI 53706, U.S.A.

Corresponding Author: Tian Zhang

- Postal Address: 555 Bailey Avenue, IBM-Santa Teresa Lab., J15/C265, San Jose, CA95141, U.S.A.

- Phone: 408-463-4106

- Fax: 408-463-3834

- Email: tian_zhang@vnet.ibm.com

Abstract. Data clustering is an important technique for exploratory data analysis, and has been studied for several years. It has been shown to be useful in many practical domains such as data classification and image processing. Recently, there has been a growing emphasis on exploratory analysis of *very large* datasets to discover useful patterns and/or correlations among attributes. This is called *data mining*, and data clustering is regarded as a particular branch. However existing data clustering methods do not adequately address the problem of processing large datasets with a limited amount of resources (e.g., memory and cpu cycles). So as the dataset size increases, they do not scale up well in terms of memory requirement, running time, and result quality.

In this paper, an efficient and scalable data clustering method is proposed, based on a new in-memory data structure called *CF-tree*, which serves as an in-memory summary of the data distribution. We have implemented it in a system called BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies), and studied its performance extensively in terms of memory requirements, running time, clustering quality, stability and scalability; we also compare it with other available methods. Finally, BIRCH is applied to solve two real-life problems: one is building an iterative and interactive pixel classification tool, and the other is generating the initial codebook for image compression.

Keywords: Very Large Databases, Data Clustering, Incremental Algorithm, Data Classification and Compression

1. Introduction

In this paper, *data clustering* refers to the problem of dividing N data points into K groups so as to minimize an intra-group 'difference' metric, such as the sum of the squared distances from the cluster centers. Given a very large set of multi-dimensional data points, the data space is usually not uniformly occupied by the data points. Through data clustering, one can identify sparse and crowded regions, and hence discover the overall distribution patterns or the correlations among data attributes. This information may be used to guide the application of more rigorous data analysis procedures. It is a problem with many practical applications and has been studied for many years. Many clustering methods have been developed and applied to various domains, including data classification and image compression [22, 5]. However, it is also a very difficult subject because theoretically, it is a *nonconvex, discrete optimization* [16] problem. Due to an abundance of local minima, there is typically no way to find a globally minimal solution without trying all possible partitions. Usually, this is infeasible except when N and K are extremely small.

In this paper, we add the following database-oriented constraints to the problem, motivated by our desire to cluster very large datasets: *The amount of memory available is limited (typically, much smaller than the dataset size) whereas the dataset can be arbitrarily large, and the I/O cost involved in clustering the dataset should be minimized.* We present a clustering algorithm called BIRCH and demonstrate that it is especially suitable for clustering very large datasets. BIRCH deals with large datasets by first generating a more compact summary that retains as much distribution information as possible, and then clustering the data summary instead of the original dataset. Its I/O cost is linear with the the dataset size: a *single*

scan of the dataset yields a good clustering, and one or more additional passes can (optionally) be used to improve the quality further.

By evaluating BIRCH's running time, memory usage, clustering quality, stability and scalability, as well as comparing it with other existing algorithms, we argue that BIRCH is the best available clustering method for handling very large datasets. We note that BIRCH actually complements other clustering algorithms by virtue of the fact that different clustering algorithms can be applied to the summary produced by BIRCH. BIRCH's architecture also offers opportunities for parallel and concurrent clustering, and it is possible to interactively and dynamically tune the performance based on knowledge gained about the dataset over the course of the execution.

The rest of the paper is organized as follows. Section 2 surveys related work, and then explains the contributions and limitations of BIRCH. Section 3 presents some background material needed for discussing data clustering in BIRCH. Section 4 introduces the *clustering feature* (CF) concept and the *CF-tree* data structure, which are central to BIRCH. The details of the BIRCH data clustering algorithm are described in Section 5. The performance study of BIRCH, CLARANS and KMEANS on synthetic datasets is presented in Section 6. Section 7 presents two applications of BIRCH, which are also intended to show how BIRCH, CLARANS and KMEANS perform on some real datasets. Finally our conclusions and directions for future research are presented in Section 8.

2. Previous Work and BIRCH

Data clustering has been studied in the Machine Learning [2, 9, 10, 12, 21], Statistics [4, 5, 22, 23], and Database [6, 7, 24] communities with different methods and different emphases.

2.1. Probability-Based Clustering

Previous data clustering work in Machine Learning is usually referred to as unsupervised conceptual learning [2, 9, 12, 21]. They concentrate on *incremental* approaches that accept instances one at a time, and do not extensively reprocess previously encountered instances while incorporating a new concept. *Concept* (or cluster) formation is accomplished by top-down 'sorting', with each new instance directed through a *hierarchy* whose nodes are formed gradually, and represent concepts. They are usually *probability-based* approaches, i.e., (1) they use probabilistic measurements (e.g., *category utility* as discussed in [9, 12]) for making decisions; and (2) they represent concepts (or clusters) with probabilistic descriptions.

For example, COBWEB [9] proceeds as follows. To insert a new instance into the hierarchy, it starts from the root, and considers four choices at each level as it descends the hierarchy: (1) recursively incorporating the instance into an existing node, (2) creating a new node for the instance, (3) merging two nodes to host the instance, and (4) splitting an existing node to host the instance. The choice that

results in the highest *category utility* score is selected. COBWEB has the following limitations:

- It is targeted for handling discrete attributes and the category utility measurement used is very expensive to compute. To compute the category utility scores, a *discrete probability distribution* is stored in each node for *each individual attribute*. COBWEB makes the assumption that probability distributions on separate attributes are statistically independent, and ignores correlations among attributes. Updating and storing a concept is very expensive, especially if the attributes have a large number of values. COBWEB deals only with discrete attributes, and for a continuous attribute, one has to divide the attribute values into ranges, or 'discretize' the attribute in advance.

- All instances ever encountered are retained as terminal nodes in the hierarchy. For very large datasets, storing and manipulating such a large hierarchy is infeasible. It has also been shown that this kind of large hierarchy tends to 'overfit' the data. A related problem is that this hierarchy is not kept width-balanced or height-balanced. So in the case of skewed input data, this may cause performance to degrade.

Another system called CLASSIT [12] is very similar to COBWEB, with the following main differences. (1) It only deals with *continuous* (or real-valued) attributes (in contrast to discrete attributes in COBWEB). (2) It stores a *continuous normal distribution* (i.e., mean and standard deviation) for *each individual attribute* in a node, in contrast to a discrete probability distribution in COBWEB. (3) As it classifies a new instance, it can halt at some higher-level node if the instance is 'similar enough' to the node, whereas COBWEB always descends to a terminal node. (4) It modifies the category utility measurement to be an integral over continuous attributes, instead of a sum over discrete attributes as in COBWEB.

The disadvantages of using an expensive metric and generating large, unbalanced tree structures clearly apply to CLASSIT as well as COBWEB, and make it unsuitable for working directly with large datasets.

2.2. Distance-Based

Most data clustering algorithms in Statistics are distance-based approaches. That is, (1) they assume that there is a distance measurement between any two instances (or data points), and that this measurement can be used for making similarity decisions; and (2) they represent clusters by some kind of 'center' measure.

There are two categories of clustering algorithms [16]: *Partitioning Clustering* and *Hierarchical Clustering*. Partitioning Clustering (PC) [4, 16] starts with an initial partition, then tries all possible moving or swapping of data points from one group to another iteratively to optimize the objective measurement function. Each cluster is represented either by the *centroid* of the cluster (KMEANS), or by one object centrally located in the cluster (KMEDOIDS). It guarantees convergence

to a local minimum, but the quality of the local minimum is very sensitive to the initial partition, and the worst case time complexity is exponential. Hierarchical Clustering (HC) [4, 23] does not try to find the 'best' clusters, instead it keeps merging (agglomerative HC) the closest pair, or splitting (divisive HC) the farthest pair, of objects to form clusters. With a reasonable distance measurement, the best time complexity of a practical HC algorithm is $O(N^2)$.

In summary, these approaches assume that all data points are given in advance and can be stored in memory and scanned frequently (non-incremental). They totally or partially ignore the fact that not all data points in the dataset are equally important for purposes of clustering, i.e., that data points which are close and dense can be considered collectively instead of individually. They are *global* or *semi-global* methods at the granularity of data points. That is, for each clustering decision, they inspect all data points or all currently existing clusters equally no matter how close or far away they are, and they use global measurements, which require scanning all data points or all currently existing clusters. Hence none of them can scale up linearly with stable quality.

Data clustering has been recognized as a useful spatial data mining method recently. [24] presents CLARANS, which is a KMEDOIDS algorithm but with randomized partial search strategy, and suggests that CLARANS out-performs the traditional KMEDOIDS algorithms. The clustering process in CLARANS is formalized as searching a graph in which each node is a K-partition represented by K medoids, and two nodes are neighbors if they only differ by one medoid. CLARANS starts with a randomly selected node. For the current node, it checks at most *maxneighbor* neighbors randomly, and if a better neighbor is found, it moves to the neighbor and continues; otherwise it records the current node as a *local minimum*, and restarts with a new randomly selected node to search for another *local minimum*. CLARANS stops after *numlocal local minima* have been found, and returns the best of these. CLARANS suffers from the same drawbacks as the KMEDOIDS method with respect to efficiency. In addition, it may not find a real local minimum due to the random search trimming controlled by *maxneighbor*.

The R-tree [13] (or variants such as R^*-tree [1]) is a popular dynamic multi-dimensional spatial index structure that has existed in the database community for more than a decade. Based on spatial locality in R^*-trees (a variation of R-trees), [6] and [7] propose focusing techniques to improve CLARANS's ability to deal with very large datasets that may reside on disks by (1) clustering a sample of the dataset that is drawn from each R^*-tree data page; and (2) focusing on relevant data points for distance and quality updates. Their experiments show that the time is improved but with a small loss of quality.

2.3. Contributions and Limitations of BIRCH

The CF-tree structure introduced in this paper is strongly influenced by balanced tree-structured indexes such as B-trees and R-Trees. It is also influenced by the incremental and hierarchical themes of COBWEB as well as COBWEB's use of

splitting and merging to alleviate the potential sensitivity to input data ordering. Currently, BIRCH can only deal with *metric* attributes (similar to the kind of attributes that KMEANS and CLASSIT can handle). A *metric* attribute is one whose values can be represented by explicit coordinates in an *Euclidean* space.

In contrast to earlier work, an important contribution of BIRCH is the *formulation* of the clustering problem in a way that is appropriate for very large datasets by making the time and memory constraints explicit. Another contribution is that BIRCH exploits the observation that the data space is usually not uniformly occupied, and hence not every data point is equally important for clustering purposes. So BIRCH treats a dense region of points (or a subclusters) collectively by storing a compact *summarization* ('clustering feature', which is discussed in Section 4.1). BIRCH thereby reduces the problem of clustering the original data points into one of clustering the set of summaries, which is much smaller than the original dataset.

The summaries generated by BIRCH reflect the natural *closeness* of data, allow for the computation of the distance-based measurements (defined in Section 3), and can be maintained efficiently and incrementally. Although we only use them for computing the distances, we note that they are also sufficient for computing the probability-based measurements such as mean, standard deviation and category utility used in CLASSIT.

Compared with prior distance-based algorithms, BIRCH is *incremental* in the sense that clustering decisions are made without scanning all data points or all currently existing clusters. If we omit the optional Phase 4 (Section 5), BIRCH is an incremental method that does not require the whole dataset in advance, and only scans the dataset once.

Compared with prior probability-based algorithms, BIRCH tries to make the best use of the available memory to derive the finest possible subclusters (to ensure accuracy) while minimizing I/O costs (to ensure efficiency) by organizing the clustering and reducing process using an in-memory *balanced* tree structure of bounded size. Finally BIRCH does not assume that the probability distributions on separate attributes are independent.

3. Background

Assuming that the readers are familiar with the terminology of vector spaces, we begin by defining centroid, radius and diameter for a cluster. Given N d-dimensional data points in a cluster: $\{\vec{X}_i\}$ where $i = 1, 2, ..., N$, the **centroid** $\vec{X0}$, **radius** R and **diameter** D of the cluster are defined as:

$$\vec{X0} = \frac{\sum_{i=1}^{N} \vec{X}_i}{N} \tag{1}$$

$$R = \left(\frac{\sum_{i=1}^{N} (\vec{X}_i - \vec{X0})^2}{N}\right)^{\frac{1}{2}} \tag{2}$$

$$D = \left(\frac{\sum_{i=1}^{N}\sum_{j=1}^{N}(\vec{X_i}-\vec{X_j})^2}{N(N-1)}\right)^{\frac{1}{2}} \tag{3}$$

R is the average distance from member points to the centroid. D is the average pairwise distance within a cluster. They are two alternative measures of the tightness of the cluster around the centroid. Next, between two clusters, we define five alternative distances for measuring their closeness.

Given the centroids of two clusters: $\vec{X0}_1$ and $\vec{X0}_2$, the **centroid Euclidean distance** $D0$ and **centroid Manhattan distance** $D1$ of the two clusters are defined as:

$$D0 = ((\vec{X0}_1 - \vec{X0}_2)^2)^{\frac{1}{2}} \tag{4}$$

$$D1 = |\vec{X0}_1 - \vec{X0}_2| = \sum_{i=1}^{d} |\vec{X0}_1{}^{(i)} - \vec{X0}_2{}^{(i)}| \tag{5}$$

Given N_1 d-dimensional data points in a cluster: $\{\vec{X_i}\}$ where $i = 1, 2, ..., N_1$, and N_2 data points in another cluster: $\{\vec{X_j}\}$ where $j = N_1+1, N_1+2, ..., N_1+N_2$, the **average inter-cluster distance** $D2$, **average intra-cluster distance** $D3$ and **variance increase distance** $D4$ of the two clusters are defined as:

$$D2 = \left(\frac{\sum_{i=1}^{N_1}\sum_{j=N_1+1}^{N_1+N_2}(\vec{X_i}-\vec{X_j})^2}{N_1 N_2}\right)^{\frac{1}{2}} \tag{6}$$

$$D3 = \left(\frac{\sum_{i=1}^{N_1+N_2}\sum_{j=1}^{N_1+N_2}(\vec{X_i}-\vec{X_j})^2}{(N_1+N_2)(N_1+N_2-1)}\right)^{\frac{1}{2}} \tag{7}$$

$$D4 = \left(\sum_{k=1}^{N_1+N_2}(\vec{X_k} - \frac{\sum_{l=1}^{N_1+N_2}\vec{X_l}}{N_1+N_2})^2 - \sum_{i=1}^{N_1}(\vec{X_i} - \frac{\sum_{l=1}^{N_1}\vec{X_l}}{N_1})^2 - \sum_{j=N_1+1}^{N_1+N_2}(\vec{X_j} - \frac{\sum_{l=N_1+1}^{N_1+N_2}\vec{X_l}}{N_2})^2\right)^{\frac{1}{2}} \tag{8}$$

$D3$ is actually D of the merged cluster. For the sake of clarity, we treat $\vec{X0}$, R and D as properties of a single cluster, and $D0$, $D1$, $D2$, $D3$ and $D4$ as properties between two clusters and state them separately.

Following are two alternative clustering quality measurements: **weighted average cluster radius square** $Q1$ (or $\bar{R}$), and **weighted average cluster diameter square** $Q2$ (or $\bar{D}$).

$$Q1 = \frac{\sum_{i=1}^{K} n_i R_i^2}{\sum_{i=1}^{K} n_i} \tag{9}$$

$$Q2 = \frac{\sum_{i=1}^{K} n_i(n_i-1)D_i^2}{\sum_{i=1}^{K} n_i(n_i-1)} \tag{10}$$

We can optionally pre-process data by weighting and/or shifting the data along different dimensions without affecting the relative placement of data points (That is if point A is to the left of point B, then after weighting and shifting, point A is still to the left of point B.). For example, to normalize the data, one can shift it by the mean value along each dimension, and then weight it by the inverse of the standard deviation on each dimension. In general, such data pre-processing is a debatable semantic issue. On one hand, it avoids biases caused by some dimensions. For example, the dimensions with large spread dominate the distance calculations in the clustering process. On the other hand, it is inappropriate if the spread is indeed due to natural differences of clusters. Since pre-processing the data in such a manner is orthogonal to the clustering algorithm itself, we will assume that the user is responsible for such pre-processing, and not consider it further.

4. Clustering Feature and CF-tree

BIRCH summarizes a dataset into a set of subclusters to reduce the scale of the clustering problem. In this section, we will answer the following questions about the summarization used in BIRCH:

1. How much information should be kept for each subcluster?
2. How is the information about subclusters organized?
3. How efficiently is the organization maintained?

4.1. Clustering Feature (CF)

A **Clustering Feature** (CF) entry is a triple summarizing the information that we maintain about a subcluster of data points.

CF Definition *: Given N d-dimensional data points in a cluster: $\{\vec{X}_i\}$ where $i = 1, 2, ..., N$, the* **Clustering Feature (CF)** *entry of the cluster is defined as a triple:* $\mathbf{CF} = (N, \vec{LS}, SS)$*, where N is the number of data points in the cluster, $\vec{LS}$ is the linear sum of the N data points, i.e., $\sum_{i=1}^{N} \vec{X}_i$, and SS is the square sum of the N data points, i.e., $\sum_{i=1}^{N} \vec{X}_i^{\,2}$.*

CF Representativity Theorem *: Given the CF entries of subclusters, all the measurements defined in Section 3 can be computed accurately.*

CF Additivity Theorem *: Assume that* $\mathbf{CF_1} = (N_1, \vec{LS}_1, SS_1)$*, and* $\mathbf{CF_2} = (N_2, \vec{LS}_2, SS_2)$ *are the CF entries of two disjoint subclusters. Then the CF entry of the subcluster that is formed by merging the two disjoint subclusters is:*

$$\mathbf{CF_1} + \mathbf{CF_2} = (N_1 + N_2, \vec{LS}_1 + \vec{LS}_2, SS_1 + SS_2) \tag{11}$$

The theorem's proof consists of conventional vector space algebra [28]. According to the CF definition and the CF representativity theorem, one can think of a subcluster as a set of data points, and the CF entry stored as a summary. This CF entry is not only compact because it stores much less than all the data points in the subcluster, it is also accurate because it is sufficient for calculating all the measurements (as defined in Section 3) that we need for making clustering decisions in BIRCH. According to the CF additivity theorem, the CF entries can be stored and calculated incrementally and consistently as subclusters are merged or new data points are inserted.

4.2. CF-tree

A CF-tree is a height-balanced tree with two parameters: branching factor (B for nonleaf node and L for leaf node) and threshold T. Each nonleaf node contains at most B entries of the form $[CF_i, child_i]$, where $i = 1, 2, ..., B$, '$child_i$' is a pointer to its i-th child node, and CF_i is the CF entry of the subcluster represented by this child. So a nonleaf node represents a subcluster made up of all the subclusters represented by its entries. A leaf node contains at most L entries, and each entry is a CF. In addition, each leaf node has two pointers, '$prev$' and '$next$', which are used to chain all leaf nodes together for efficient scans. A leaf node also represents a subcluster made up of all the subclusters represented by its entries. But all entries in a leaf node must satisfy a *threshold requirement*, with respect to a threshold value T: *the diameter (alternatively, the radius) of each leaf entry has to be less than* T.

The tree size is a function of T. The larger T is, the smaller the tree is. We require a node to fit in a page of size P, where P is a parameter of BIRCH. Once the dimension d of the data space is given, the sizes of leaf and nonleaf entries are known, and then B and L are determined by P. So P can be varied for performance tuning.

Such a CF-tree will be built dynamically as new data objects are inserted. It is used to guide a new insertion into the correct subcluster for clustering purposes just as a B+-tree is used to guide a new insertion into the correct position for sorting purposes. However the CF-tree is a very compact representation of the dataset because each entry in a leaf node is not a single data point but a subcluster (which absorbs as many data points as the specific threshold value allows).

4.3. Insertion Algorithm

We now present the algorithm for inserting a CF entry 'Ent' (a single data point or a subcluster) into a CF-tree.

1. *Identifying the appropriate leaf:* Starting from the root, recursively descend the CF-tree by choosing the **closest** child node according to a chosen distance metric: $D0$,$D1$,$D2$,$D3$ or $D4$ as defined in Section 3.

2. *Modifying the leaf:* Upon reaching a leaf node, find the closest leaf entry, say L_i, and then test whether L_i can 'absorb' 'Ent' without violating the threshold condition. (That is, the cluster merged with 'Ent' and L_i must satisfy the threshold condition. Note that the CF entry of the new cluster can be computed from the CF entries for L_i and 'Ent'.) If so, update the CF entry for L_i to reflect this. If not, add a new entry for 'Ent' to the leaf. If there is space on the leaf for this new entry to **fit in**, we are done, otherwise we must *split* the leaf node. Node splitting is done by choosing the **farthest** pair of entries as seeds, and redistributing the remaining entries based on the **closest** criteria.

3. *Modifying the path to the leaf:* After inserting 'Ent' into a leaf, update the CF information for each nonleaf entry on the path to the leaf. In the absence of a split, this simply involves updating existing CF entries to reflect the addition of 'Ent'. A leaf split requires us to insert a new nonleaf entry into the parent node, to describe the newly created leaf. If the parent has space for this entry, at all higher levels, we only need to update the CF entries to reflect the addition of 'Ent'. In general, however, we may have to split the parent as well, and so on up to the root. If the root is split, the tree height increases by one.

4. *A Merging Refinement:* Splits are caused by the page size, which is independent of the clustering properties of the data. In the presence of skewed data input order , this can affect the clustering quality, and also reduce space utilization. A simple additional merging step often helps ameliorate these problems: Suppose that there is a leaf split, and the propagation of this split stops at some nonleaf node N_j, i.e., N_j can accommodate the additional entry resulting from the split. We now scan node N_j to find the two **closest** entries. If they are not the pair corresponding to the split, we try to merge them and the corresponding two child nodes. If there are more entries in the two child nodes than one page can hold, we split the merging result again. During the resplitting, in case one of the seeds attracts enough merged entries to fill a page, we just put the rest of the entries with the other seed. In summary, if the merged entries fit on a single page, we free a node (page) for later use and create space for one more entry in node N_j, thereby increasing space utilization and postponing future splits; otherwise we improve the distribution of entries in the closest two children.

The above steps work together to dynamically adjust the CF-tree to reduce its sensitivity to the data input ordering.

4.4. Anomalies

Since each node can only hold a limited number of entries due to its fixed size, it does not always correspond to a natural cluster. Occasionally, two subclusters that should have been in one cluster are split across nodes. Depending upon the order of data input and the degree of skew, it is also possible that two subclusters that should not be in one cluster are kept in the same node. This infrequent but

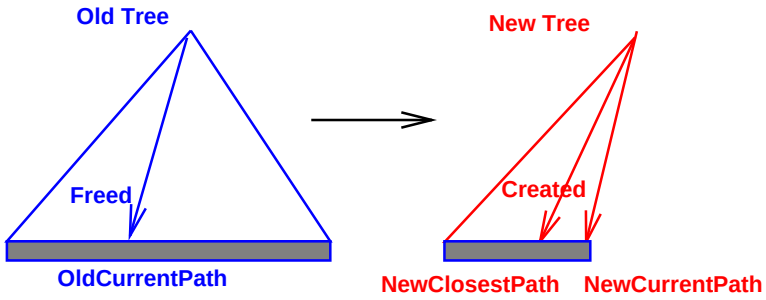


Figure 1. Rebuilding CF-tree

undesirable anomaly caused by node size limit (Anomaly 1) will be addressed with a global clustering algorithm discussed in Section 5.

Another undesirable artifact is that if the same data point is inserted twice, but at different times, the two copies might be entered into two distinct leaf entries. In other words, occasionally with a skewed input order, a point might enter a leaf entry that it should not have entered (Anomaly 2). This problem will be addressed with a refining algorithm discussed in Section 5.

4.5. Rebuilding Algorithm

We now discuss how to rebuild the CF-tree by increasing the threshold if the CF-tree size limit is exceeded as data points are inserted. Assume that t_i is a CF-tree of threshold T_i. Its height is h, and its size (number of nodes) is S_i. Given $T_{i+1} \geq T_i$, we want to use all the leaf entries of t_i to rebuild a CF-tree, t_{i+1}, of threshold T_{i+1} such that the size of t_{i+1} should not be larger than S_i.

Assume that within each node of CF-tree t_i, the entries are labeled contiguously from 0 to $n_k - 1$, where n_k is the number of entries in that node. Then a **path** from an entry in the root (Level 1) to a leaf node (Level h) can be uniquely represented by $(i_1, i_2, ..., i_{h-1})$, where $i_j, j = 1, ..., h-1$ is the label of the j-th level entry on that path. So naturally, path $(i_1^{(1)}, i_2^{(1)}, ..., i_{h-1}^{(1)})$ is **before (or $<$)** path $(i_1^{(2)}, i_2^{(2)}, ..., i_{h-1}^{(2)})$ if $i_1^{(1)} = i_1^{(2)}$, ..., $i_{j-1}^{(1)} = i_{j-1}^{(2)}$, and $i_j^{(1)} < i_j^{(2)} (0 \leq j \leq h-1)$. It is obvious that each leaf node corresponds to a path, since we are dealing with tree structures, and we will just use 'path' and 'leaf node' interchangeably from now on.

The idea of the rebuilding algorithm is illustrated in Figure 1. With the natural path order defined above, it scans and frees the old tree, path by path, and at the same time, creates the new tree path by path. The new tree starts with NULL, and 'OldCurrentPath' is initially the leftmost path in the old tree.

1. *Create the corresponding 'NewCurrentPath' in the new tree*: Copy the nodes along 'OldCurrentPath' in the old tree into the new tree as the (current) rightmost path; call this 'NewCurrentPath'.

2. *Insert leaf entries in 'OldCurrentPath' to the new tree*: With the new threshold, each leaf entry in 'OldCurrentPath' is tested against the new tree to see if it can either be **absorbed** by an existing leaf entry, or **fit in** as a new leaf entry without splitting, in the 'NewClosestPath' that is found top-down with the **closest** criteria in the new tree. If yes and 'NewClosestPath' is **before** 'NewCurrentPath', then it is inserted to 'NewClosestPath', and deleted from the leaf node in 'NewCurrentPath'.

3. *Free space in 'OldCurrentPath' and 'NewCurrentPath'*: Once all leaf entries in 'OldCurrentPath' are processed, the nodes along 'OldCurrentPath' can be deleted from the old tree. It is also likely that some nodes along 'NewCurrentPath' are empty because leaf entries that originally corresponded to this path have been 'pushed forward'. In this case the empty nodes can be deleted from the new tree.

4. *Process the next path in the old tree:* 'OldCurrentPath' is set to the next path in the old tree if there still exists one, and the above steps are repeated.

From the rebuilding steps, it is clear that all leaf entries in the old tree are reinserted into the new tree, but the new tree can never become larger than the old tree. Since only nodes corresponding to 'OldCurrentPath' and 'NewCurrentPath' need to exist simultaneously in both trees, the maximum extra space needed for the tree transformation is h (height of the old tree) pages. So by increasing the threshold value T, we can rebuild a smaller CF-tree with a very limited amount of extra memory. The following theorem summarizes these observations.

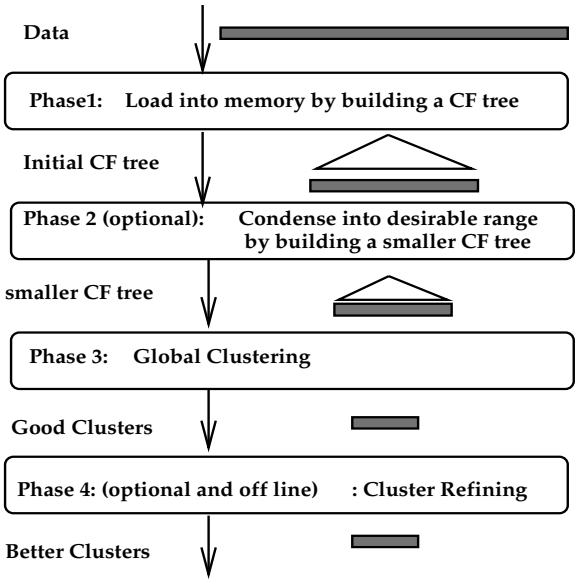
Reducibility Theorem *: Assume we rebuild CF-tree t_{i+1} of threshold T_{i+1} from CF-tree t_i of threshold T_i by the above algorithm, and let S_i and S_{i+1} be the sizes of t_i and t_{i+1} respectively. If $T_{i+1} \geq T_i$, then $S_{i+1} \leq S_i$, and the transformation from t_i to t_{i+1} needs at most h extra pages of memory, where h is the height of t_i.*

5. The BIRCH Clustering Algorithm

Figure 2 presents the overview of BIRCH. It consists of four phases: (1) Loading, (2) Optional Condensing, (3) Global Clustering, and (4) Optional Refining.

The main task of Phase 1 is to scan all data and build an initial in-memory CF-tree using the given amount of memory and recycling space on disk. This CF-tree tries to reflect the clustering information of the dataset in as much detail as possible subject to the memory limits. With crowded data points grouped into subclusters, and sparse data points removed as outliers, this phase creates an in-memory summary of the data. More details of Phase 1 will be discussed in Section 5.1. After Phase 1, subsequent computations in later phases will be: (1) fast because (a) no I/O operations are needed, and (b) the problem of clustering the original data is reduced to a smaller problem of clustering the subclusters in the leaf entries; (2) accurate because (a) outliers can be eliminated, and (b) the remaining

Figure 2. BIRCH Overview



data is described at the finest granularity that can be achieved given the available memory; (3) less order sensitive because the leaf entries of the initial tree form an input order containing better data locality compared with the arbitrary original data input order.

Once all the clustering information is loaded into the in-memory CF-tree, we can use an existing global or semi-global algorithm in Phase 3 to cluster all the leaf entries across the boundaries of different nodes. This way we can overcome Anomaly 1, (Section 4.4) which causes the CF-tree nodes to be unfaithful to the actual clusters in the data. We observe that existing clustering algorithms (e.g., HC, KMEANS and CLARANS) that work with a set of data points can be readily adapted to work with a set of subclusters, each described by its CF entry.

We adapted an agglomerative hierarchical clustering algorithm based on the description in [25]. It is applied to the subclusters represented by their CF entries. It has a complexity of $O(m^2)$, where m is the number of subclusters. If the distance metric satisfies the *reducibility property* [25][1], it produces exact results; otherwise it still provides a very good approximate algorithm. In our case, $D2$ and $D4$ satisfy the reducibility property and they are the ideal metrics for using this algorithm. Besides, it has the flexibility of allowing the user to explore different number of clusters (K), or different diameter (or radius) thresholds for clusters (T) based on the formed hierarchy without re-scanning the data or re-clustering the subclusters.

Phase 2 is an optional phase. With experimentation, we have observed that the global or semi-global clustering methods that we adapt in Phase 3 have different

input size ranges within which they perform well in terms of both speed and quality. For example, if we choose to adapt CLARANS in Phase 3, we know that CLARANS performs pretty well for a set of less than 5000 data objects. That is because within that range, frequent data scanning is acceptable and getting trapped at a very bad local minimal due to the partial searching is not very likely. So potentially there is a gap between the size of Phase 1 results and the best performance range of the Phase 3 algorithm we select. Phase 2 serves as a cushion between Phase 1 and Phase 3 and bridges this gap: we scan the leaf entries in the initial CF-tree to rebuild a smaller CF-tree, while removing more outliers and grouping more crowded subclusters into larger ones.

After Phase 3, we obtain a set of clusters that captures the major distribution patterns in the data. However, minor and localized inaccuracies might exist because of (1) the rare misplacement problem (Anomaly 2 in Section 4.4), and (2) the fact that Phase 3 is applied on a coarse summary of the data. Phase 4 is optional and entails the cost of additional passes over the data to correct those inaccuracies and refine the clusters further. Note that up to this point, the original data has only been scanned once, although the tree may have been rebuilt multiple times.

Phase 4 uses the centroids of the clusters produced by Phase 3 as seeds, and redistributes the data points to its closest seed to obtain a set of new clusters. Not only does this allow points belonging to a cluster to migrate, but also it ensures that all copies of a given data point go to the same cluster. Phase 4 can be extended with additional passes if desired by the user, and it has been proved to converge to a minimum [11]. As a bonus, during this pass, each data point can be labeled with the cluster that it belongs to, if we wish to identify the data points in each cluster. Phase 4 also provides us with the option of discarding outliers. That is, a point which is too far from its closest seed can be treated as an outlier and not included in the result.

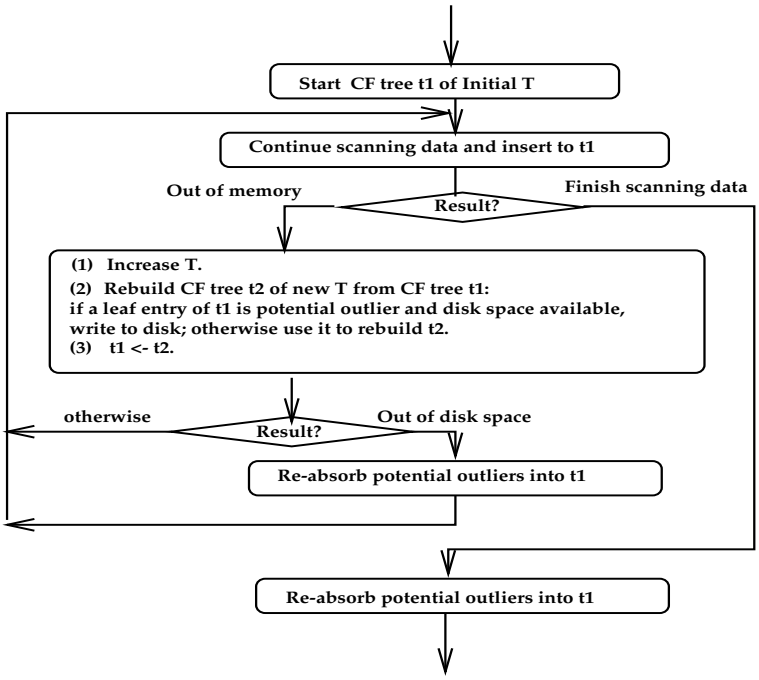
5.1. Phase 1 Revisited

Figure 3 shows the details of Phase 1. It starts with an initial threshold value, scans the data, and inserts points into the tree. If it runs out of memory before it finishes scanning the data, it increases the threshold value, and rebuilds a new, *smaller* CF-tree, by re-inserting the leaf entries of the old CF-tree into the new CF-tree. After all the old leaf entries have been re-inserted, the scanning of the data (and insertion into the new CF-tree) is resumed from the point at which it was interrupted.

5.1.1. Threshold Heuristic

A good choice of threshold value can greatly reduce the number of rebuilds. Since the initial threshold value T_0 is increased dynamically, we can adjust for its being too low. But if the initial T_0 is too high, we will obtain a less detailed CF-tree than

Figure 3. Flow Chart of Phase 1



is feasible with the available memory. So T_0 should be set conservatively. BIRCH sets it to zero by default; a knowledgeable user could change this.

Suppose that T_i turns out to be too small, and we subsequently run out of memory after N_i data points have been scanned. Based on the portion of the data that we have scanned and the CF-tree that we have built up so far, we try to estimate the next threshold value T_{i+1}. This estimation is a difficult problem, and a full solution is beyond the scope of this paper. Currently, we use the following memory-utilization oriented heuristic approach: If the current CF-tree occupies all the memory, we increase the threshold value to be the average of the distances between all the 'nearest pairs' of leaf entries. So on average, approximately two leaf entries will be merged into one with the new threshold value. For efficiency reasons, the distance of each 'nearest pair' of leaf entries is approximated by searching only within the same leaf node locally (instead of searching all the leaf entries globally). With the CF-tree insertion algorithm, it is very likely that the nearest neighbor of a leaf entry is within the same leaf node as that leaf entry is.

There are several advantages of estimating the threshold value this way: (1) the new CF-tree, which is rebuilt from the current CF-tree, will occupy approximately half of the memory, leaving the other half for accommodating additional incoming data points. So no matter how the incoming data is distributed or is input, the

memory utilization is always maintained approximately at 50%; and (2) only the distribution of the seen data, which is stored in the current CF-tree, is needed in the estimation. However, more sophisticated solutions to the threshold estimation problem should be studied in the future.

5.1.2. Outlier-Handling Option

Optionally, we can allocate R bytes of disk space for handling *outliers*. Outliers are leaf entries of low density that are judged to be unimportant with respect to the overall clustering pattern. When we rebuild the CF-tree by re-inserting the old leaf entries, the size of the new CF-tree is reduced in two ways. First, we increase the threshold value, thereby allowing each leaf entry to 'absorb' more points. Second, we treat some leaf entries as potential outliers and write them out to disk. An old leaf entry is considered to be a potential outlier if it has 'far fewer' data points than the average. 'Far fewer' is of course another heuristic.

Periodically, the disk space may run out, and the potential outliers are scanned to check if they can be re-absorbed into the current tree without causing the tree to grow in size; an increase in the threshold value or a change in the distribution due to the new data read after a potential outlier is written out could well mean that the potential outlier no longer qualifies as an outlier. When all data has been scanned, the potential outliers left in the outlier disk space must be scanned to verify if they are indeed outliers. If a potential outlier can not be absorbed at this last chance, it is very likely a real outlier and should be removed.

Note that the entire cycle — insufficient memory triggering a rebuilding of the tree, insufficient disk space triggering a re-absorbing of outliers, etc. — could be repeated several times before the dataset is fully scanned. This effort must be considered in addition to the cost of scanning the data in order to assess the cost of Phase 1 accurately.

5.1.3. Delay-Split Option

When we run out of main memory, it may well be the case that still more data points can fit in the current CF-tree without changing the threshold. However, some of the data points that we read may require us to split a node in the CF-tree, A simple idea is to write such data points to disk in a manner similar to how outliers are written, and to proceed reading the data until we run out of disk space as well. The advantage of this approach is that in general, more data points can fit in the tree before we have to rebuild.

5.2. Memory Management

We have observed that the amount of memory needed for BIRCH to find a good clustering from a given dataset is determined not by the dataset size, but by the data distribution. On the other hand, the amount of memory available to BIRCH is determined by the computing system. So it is very likely that the memory needed and the memory available do not match.

If the memory available is less than the memory needed, then BIRCH can trade running time for memory. Specifically, in Phases 1 through 3, it tries to use all the available memory to generate the subclusters that is as fine as the memory allows, but in Phase 4, by refining the clustering a few more passes, it can compensate for the inaccuracies caused by the coarseness due to insufficient memory in Phases 1.

If the memory available is more than the memory needed, then BIRCH can cluster the given dataset on multiple combinations of attributes concurrently while sharing the same scan of the dataset. So the total available memory will be divided and allocated to the clustering process of each combination of attributes accordingly. This gives the user the chance of exploring the same dataset from multiple perspectives concurrently if the available resources allow this.

6. Performance Studies

6.1. Analysis

First we analyze the cpu cost of Phase 1. Given the memory is M bytes and each page is P bytes, the maximum size of the tree is $\frac{M}{P}$. To insert a point, we need to follow a path from root to leaf, touching about $1 + \log_B \frac{M}{P}$ nodes. At each node we must examine B entries, looking for the 'closest' one; the cost per entry is proportional to the dimension d. So the cost for inserting all data points is $O(d * N * B(1 + \log_B \frac{M}{P}))$. In case we must rebuild the tree, let $C * d$ be the CF entry size where C is some constant mapping dimension into CF entry size. There are at most $\frac{M}{C*d}$ leaf entries to re-insert, so the cost of re-inserting leaf entries is $O(d*\frac{M}{C*d}*B(1+\log_B \frac{M}{P}))$. The number of times we have to re-build the tree depends upon our threshold heuristic. Currently, it is about $\log_2 \frac{N}{N_0}$, where the value 2 arises from the fact that we always bring the tree size down to about half size, and N_0 is the number of data points loaded into memory with threshold T_0. So the total cpu cost of Phase 1 is $O(d * N * B(1 + \log_B \frac{M}{P}) + \log_2 \frac{N}{N_0} * d * \frac{M}{C*d} * B(1 + \log_B \frac{M}{P}))$. Since B equals $\frac{P}{C*d}$, the total cpu cost of Phase 1 can be further rewritten as $O(N * \frac{P}{C}(1 + \log_{\frac{P}{C*d}} \frac{M}{P}) + \log_2 \frac{N}{N_0} * \frac{M}{C*d} * \frac{P}{C}(1 + \log_{\frac{P}{C*d}} \frac{M}{P}))$ The analysis of Phase 2 cpu cost is similar, and hence omitted.

As for I/O, we scan the data once in Phase 1 and not at all in Phase 2. With the outlier-handling and split-delaying options on, there is some cost associated with writing out outlier entries to disk and reading them back during a rebuilt. Considering that the amount of disk available for outlier-handling and split-delaying

Table 1. Data Generation Parameters and Their Values or Ranges Experimented

Parameter	Values or Ranges
Dimension d	2 .. 50
Pattern	grid, sine, random
Number of clusters K	4 .. 256
n_l (Lower n)	0 .. 2500
n_h (Higher n)	50 .. 2500
r_l (Lower r)	0 .. $\sqrt{50}$
r_h (Higher r)	$\sqrt{2}$ .. $\sqrt{50}$
Distance multiplier k_g	4 (grid only)
Number of cycles n_c	4 (sine only)
Noise rate r_n (%)	0 .. 10
Input order o	randomized, ordered

is not too much, and that there are about $\log_2 \frac{N}{N_0}$ rebuilds, the I/O cost of Phase 1 is not significantly different from the cost of reading in just the original dataset.

There is no I/O in Phase 3. Since the input to Phase 3 is bounded, the cpu cost of Phase 3 is therefore bounded by a constant that depends upon the maximum input size range and the global algorithm chosen for this phase. Based on the above analysis — which is actually rather pessimistic for B, number of leaf entries and the tree size in the light of our experimental results — the cost of Phases 1, 2 and 3 should scale up linearly with N.

Phase 4 scans the dataset again and puts each data point into the proper cluster; the time taken is proportional to $N * K$. However using the newest nearest neighbor techniques proposed in [14], for each of the N data point, instead of looking all K cluster centers to find the nearest one, it only looks those cluster centers that are **around** the data point. This way, Phase 4 can be improved quite a bit.

6.2. Synthetic Dataset Generator

To study the sensitivity of BIRCH to the characteristics of a wide range of input datasets, we have used a collection of synthetic datasets. The synthetic data generation is controlled by a set of parameters that are summarized in Table 1.

Each dataset consists of K clusters of d-dimensional data points. A cluster is characterized by the number of data points in it (n), its radius(r), and its center(c). n is in the range [n_l,n_h], and r is in the range [r_l,r_h]. (Note that when $n_l = n_h$ the number of points is fixed, and when $r_l = r_h$ the radius is fixed.) Once placed, the clusters cover a range of values in each dimension. We refer to these ranges as the 'overview' of the dataset.

The location of the center of each cluster is determined by the *pattern* parameter. Three patterns — *grid*, *sine*, and *random* — are currently supported by the generator. When the *grid* pattern is used, the cluster centers are placed on a $\sqrt{K} \times \sqrt{K}$ grid. The distance between the centers of neighboring clusters on the same row/column is controlled by k_g, and is set to $k_g \frac{(r_l + r_h)}{2}$. This leads to an overview of $[0, \sqrt{K} k_g \frac{r_l + r_h}{2}]$ on both dimensions. The *sine* pattern places the cluster

Table 2. BIRCH Parameters and Their Default Values

Scope	Parameter	Default Value
Global	Memory (M)	5% of dataset size
	Disk (R)	20% M
	Distance def.	D2
	Quality def.	$\bar{D}$
	Threshold def.	threshold for D
Phase1	Initial threshold	0.0
	Delay-split	on
	Page size (P)	1024 bytes
	Outlier-handling	off
Phase3	Input range	1000
	Algorithm	Adapted HC
Phase4	Refinement pass	1
	Discard-outlier	off

centers on a curve of sine function. The K clusters are divided into n_c groups, each of which is placed on a different cycle of the sine function. The x location of the center of cluster i is $2\pi i$ whereas the y location is $\frac{K}{n_c} * sine(2\pi i/(\frac{K}{n_c}))$. The overview of a sine dataset is therefore $[0,2\pi K]$ and $[-\frac{K}{n_c},+\frac{K}{n_c}]$ on the x and y directions respectively. The *random* pattern places the cluster centers randomly. The overview of the dataset is $[0,K]$ on both dimensions since the the x and y locations of the centers are both randomly distributed within the range $[0,K]$.

Once the characteristics of each cluster are determined, the data points for the cluster are generated according to a d-dimensional independent normal distribution whose mean is the center c, and whose variance in each dimension is $\frac{r^2}{d}$. Note that due to the properties of a normal distribution, the maximum distance between a point in the cluster and the center is unbounded. In other words, a point may be arbitrarily far from the cluster to which it 'belongs', according to the data generation algorithm. So a data point that belongs to cluster A may be closer to the center of cluster B than to the center of A, and we refer to such points as 'outsiders'.

In addition to the clustered data points, noise in the form of data points uniformly distributed throughout the overview of the dataset can be added to the dataset. The parameter r_n controls the percentage of data points in the dataset that are considered noise.

The placement of the data points in the dataset is controlled by the order parameter o. When the randomized option is used, the data points of all clusters and the noise are randomized throughout the entire dataset. Whereas when the ordered option is selected, the data points of a cluster are placed together, the clusters are placed in the order they are generated, and the noise is placed at the end.

6.3. Parameters and Default Setting

Table 3. Datasets Used as Base Workload

DS	Generator Setting	$\bar{D}_{int}$
1	$d = 2, grid, K = 100, n_l = n_h = 1000,$	
	$r_l = r_h = \sqrt{2}, k_g = 4, r_n = 0\%, o = randomized$	2.00
2	$d = 2, sine, K = 100, n_l = n_h = 1000,$	
	$r_l = r_h = \sqrt{2}, n_c = 4, r_n = 0\%, o = randomized$	2.00
3	$d = 2, random, K = 100, n_l = 0, n_h = 2000,$	
	$r_l = 0, r_h = 4, r_n = r_n = 0\%, o = randomized$	4.18

BIRCH is capable of working under various settings. Table 2 lists the parameters of BIRCH, their effecting scopes and their default values. Unless specified explicitly otherwise, an experiment is conducted under this default setting.

M was selected to be about 5% of the dataset size in the base workload used in our experiments. Since disk space (R) is just used for outliers, we assume that $R < M$ and set $R = 20\%$ of M. The experiments on the effects of the 5 distance metrics in the first 3 phases (Section 6.6) indicate that (1) using $D3$ in Phases 1 and 2 results in a much higher ending threshold, and hence produces clusters of poorer quality; (2) however, there is no distinctive performance difference among the others. We therefore decided to choose $D2$ as default. Following Statistics tradition, we chose 'weighted average diameter' (denoted as $\bar{D}$) as the quality metric. The smaller $\bar{D}$ is, the better the quality is.

The threshold is defined as a threshold for cluster diameter. In Phase 1, the initial threshold is set to 0, the default value. Based on a study of how page size affects performance (Section 6.6), we selected $P = 1024$. The delay-split option is used for building more compact CF-trees. The outlier-handling option is not used, for simplicity.

In Phase 3, most global algorithms can handle a few thousand objects well. So we set the input range to be 1000 as a default. We have chosen the adapted HC algorithm to use here. We decided to let Phase 4 refine the clusters only once with its discard-outlier option off, so that all data points will be counted in the quality measurement for fair comparisons with other methods.

6.4. Base Workload Performance

The first set of experiments was to evaluate the ability of BIRCH to cluster large datasets of various patterns and input orders. All the times are presented in *seconds* in this paper. Three 2-dimensional synthetic datasets, one for each pattern, were used; 2-dimensional datasets were chosen in part because they are easy to visualize. Table 3 presents the data generation settings for them. The weighted average diameters of the intended clusters $\bar{D}_{int}$ are also included in the table as rough quality indications of the datasets. Note that from now on, we refer to the clusters generated by the generator as the 'intended clusters', and the clusters identified by BIRCH as 'BIRCH clusters'.

Table 4. BIRCH Performance on Base Workload with respect to Time, $\bar{D}$, Input Order and #Scan of Data

DS	Time	$\bar{D}$	#Scan	D	Time	$\bar{D}$	#Scan
1	11.5	1.87	2	1o	13.6	1.87	2
2	10.7	1.99	2	2o	12.1	1.99	2
3	11.4	3.95	2	3o	12.2	3.99	2

Table 5. CLARANS Performance on Base Workload with respect to Time, $\bar{D}$, Input Order and #Scan of Data

DS	Time	D	#Scan	DS	Time	D	#Scan
1	932	2.10	3307	1o	794	2.11	2854
2	758	2.63	2661	2o	816	2.31	2933
3	835	3.39	2959	3o	934	3.28	3369

Figure 4 visualizes the intended clusters of DS1 by plotting a cluster as a circle whose center is the centroid, radius is the cluster radius, and label is the number of points in the cluster. The BIRCH clusters of DS1 are presented in Figure 7. We observe that the BIRCH clusters are very similar to the intended clusters in terms of location, number of points, and radii. The maximum and average distance between the centroids of an intended cluster and its corresponding BIRCH cluster are 0.17 and 0.07 respectively. The number of points in a BIRCH cluster is no more than 5% different from the corresponding intended cluster. The radii of the BIRCH clusters (ranging from 1.25 to 1.40 with an average of 1.32) are close to, those of the intended clusters (1.41). Note that all the BIRCH radii are smaller than the intended radii. This is because BIRCH assigns the 'outsiders' of an intended cluster to a proper BIRCH cluster. Similar conclusions can be reached by analyzing the visual presentations of the intended clusters and BIRCH clusters for DS2 (Figure 5 10).

As summarized in Table 4, it took BIRCH less than 15 seconds (on a DEC Pentium-pro station running Solaris) to cluster 100,000 data points of each dataset, which includes 2 scans of the dataset (about 1.16 seconds for one scan of ASCII file from disk). The pattern of the dataset had almost no impact on the clustering time. Table 4 also presents the performance results for three additional datasets — DS1o, DS2o and DS3o — which correspond to DS1, DS2 and DS3, respectively except that the parameter o of the generator is set to *ordered*. As demonstrated in Table 4, changing the order of the data points had almost no impact on the performance of BIRCH.

6.5. Comparisons of BIRCH, CLARANS and KMEANS

In this experiment, we compare the performance of BIRCH, CLARANS and KMEANS on the base workload. First of all, in CLARANS and KMEANS, the memory is assumed to be large enough to hold the whole dataset as well as some other linear size assisting data structures. So they need much more memory than BIRCH

Table 6. KMEANS Performance on Base Workload with respect to Time, $\bar{D}$, Input Order and #Scan of Data

DS	Time	$\bar{D}$	#Scan	DS	Time	$\bar{D}$	#Scan
1	43.9	2.09	289	1o	33.8	1.97	197
2	13.2	4.43	51	2o	12.7	4.20	29
3	32.9	3.66	187	3o	36.0	4.35	241

does. (Clearly, this assumption greatly favors these two algorithms in terms of the running time comparison!) Second, the CLARANS implementation was provided by Raymond Ng, and the KMEANS implementation was done by us based on the algorithm presented in [15], with the initial seeds selected randomly. We have observed that the performances of CLARANS and KMEANS are very sensitive to the random number generator used. A bad random number generator, such as the UNIX 'rand()' used in the original code of CLARANS, can generate random numbers that are not really random but sensitive to the data order, and hence make CLARANS and KMEANS's performance extremely unstable with the different input orders [?]. So to avoid this problem, we have replaced 'rand()' with a more elaborate random number generator. Third, in order for CLARANS to stop after an acceptable running time, we set its *maxneighbor* value to be the larger of 50 (instead of 250) and 1.25% of K(N-K), but no more than 100 (newly enforced upper limit recommended by Ng). Its *numlocal* value is still 2, as in [24].

Figure 8 and 9 visualize the CLARANS and KMEANS clusters for DS1. Comparing them with the intended clusters for DS1, we observe that: (1) The pattern of the location of the cluster centers is distorted. (2) The number of data points in a CLARANS or KMEANS cluster can be as many as 40% different from the number in the intended cluster. (3) The radii of CLARANS clusters varies largely from 1.15 to 1.94 with an average of 1.44 (larger than those of the intended clusters, 1.41). The radii of KMEANS clusters varies largely from 0.99 to 2.02 with an average of 1.38 (larger than those of BIRCH clusters, 1.32). Similar behavior can be observed in the visualization of CLARANS and KMEANS clusters for DS2.

Tables 5 and 6 summarize the performance of CLARANS and KMEANS. For all three datasets of the base workload, (1) They scan the dataset frequently. When running CLARANS and KMEANS experiments, all data are loaded into memory, only the first scan is from ASCII file on disk, and the remaining scans are in memory. Considering the time needed for each scan on disk (1.16 seconds per scan), CLARANS and KMEANS are much slower than BIRCH and their running times are more sensitive to the patterns of the dataset. (2) The $\bar{D}$ values for the CLARANS and KMEANS clusters are larger than those for the BIRCH clusters for DS1 and DS2. That means even through they spent a lot of time searching for a local minimal partition, the partition may not be as good as the non-minimal partition found by BIRCH. (3) The results for DS1o, DS2o, and DS3o show that if the data points are input in different orders, the time and quality of CLARANS and KMEANS clusters will change.

Figure 4. Intended Clusters of DS1

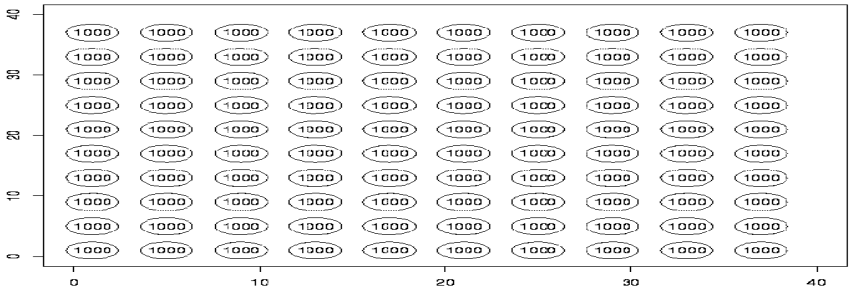


Figure 5. Intended Clusters of DS2

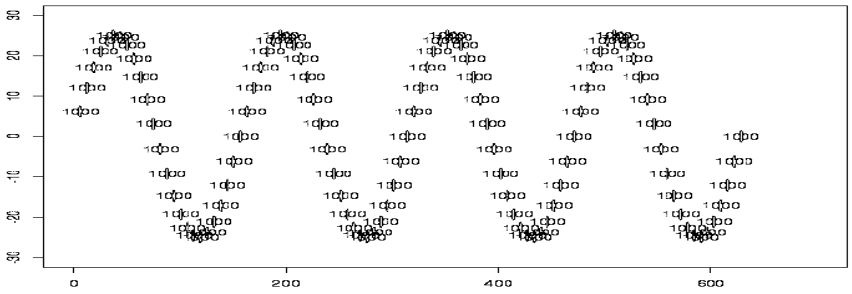
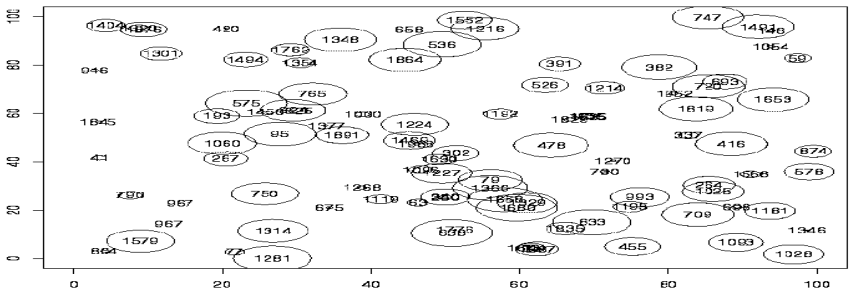


Figure 6. Intended Clusters of DS3



In conclusion, for the base workload, BIRCH uses much less memory, scans data only twice, but runs faster, is better at escaping from inferior locally minimal partitions, and less order-sensitive compared with CLARANS and KMEANS.

Figure 7. BIRCH Clusters of DS1

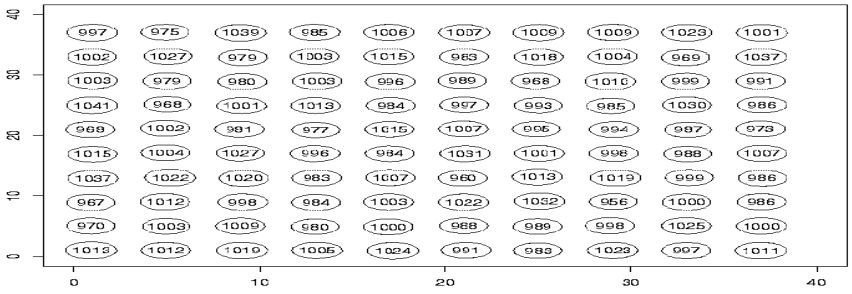


Figure 8. CLARANS Clusters of DS1

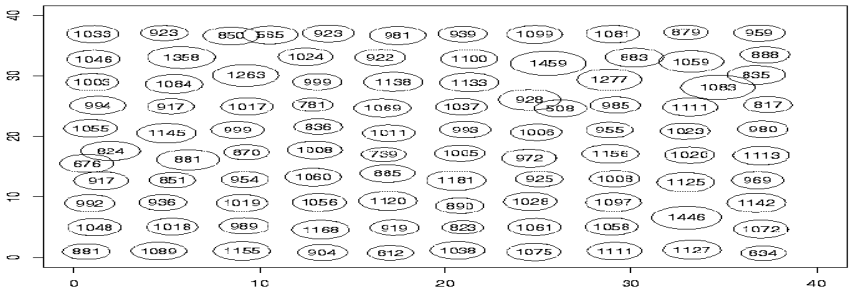
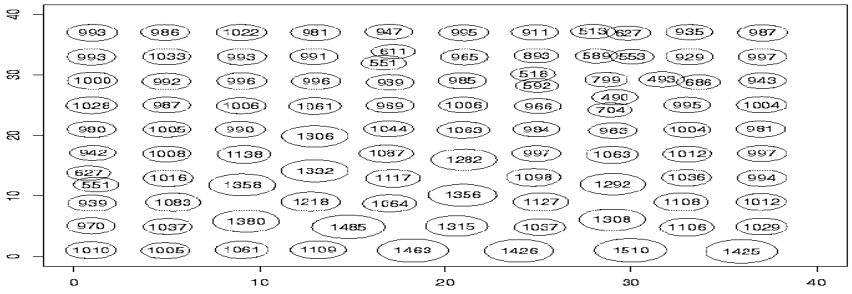


Figure 9. KMEANS Clusters of DS1



6.6. Sensitivity to Parameters

We studied the sensitivity of BIRCH's performance to several parameters. Due to lack of space, we only present some major conclusions.

Initial threshold: (1) BIRCH's performance is stable as long as the initial threshold is not excessively high with respect to the dataset. (2) $T_0 = 0.0$ works

Figure 10. BIRCH Clusters of DS2

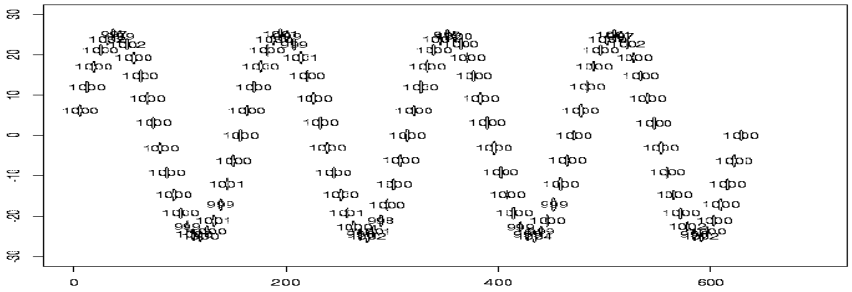


Figure 11. CLARANS Clusters of DS2

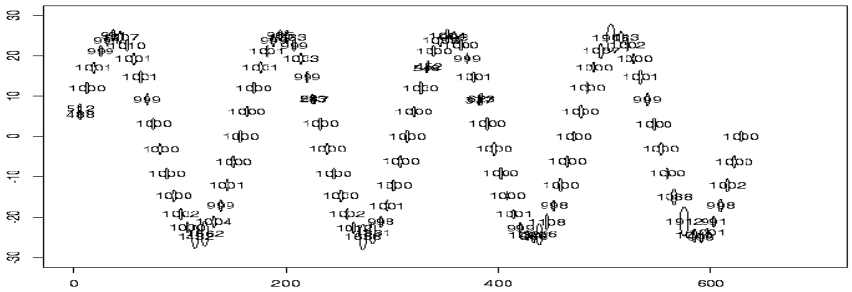
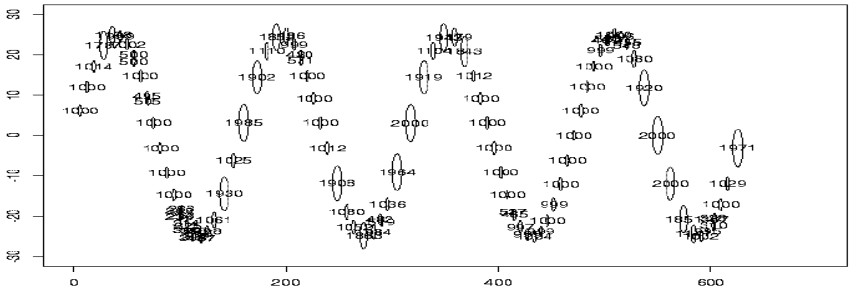


Figure 12. KMEANS Clusters of DS2



well with a little extra running time. (3) If a user does know a good T_0, then she/he can be rewarded by saving up to 10% of the time.

Page Size P: In Phase 1, smaller (larger) P tends to decrease (increase) the running time, requires higher (lower) ending threshold, produces less (more) but 'coarser (finer)' leaf entries, and hence degrades (improves) the quality. However, with the refinement in Phase 4, the experiments suggest that for $P = 256$ to 4096,

Figure 13. BIRCH Clusters of DS3

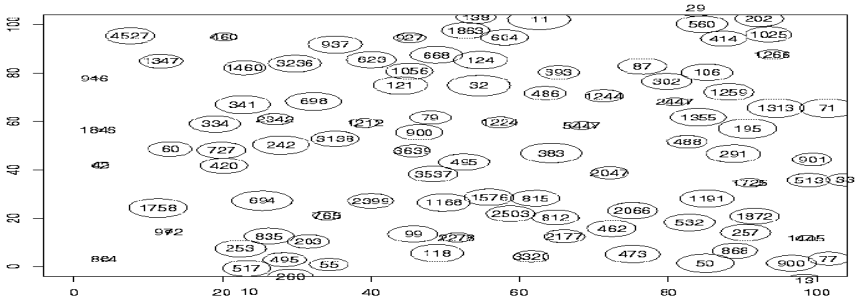


Figure 14. CLARANS Clusters of DS3

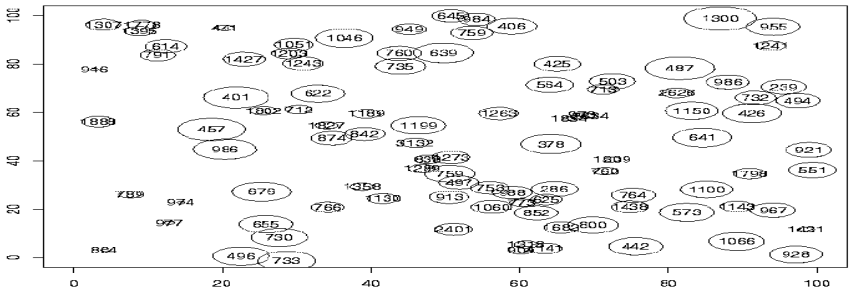
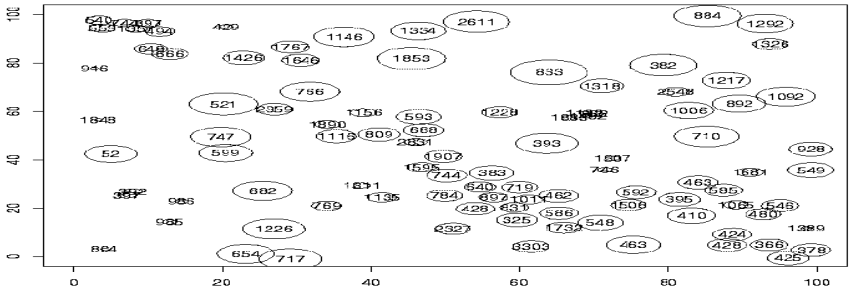


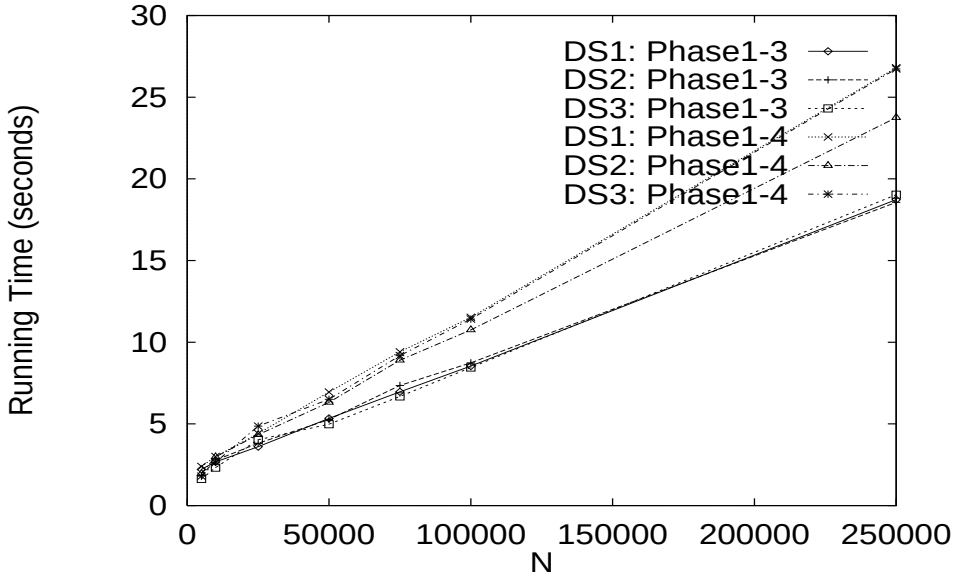
Figure 15. KMEANS Clusters of DS3



although the qualities at the end of Phase 3 are different, the final qualities after the refinement are almost the same.

Outlier Options: BIRCH was tested on 'noisy' datasets with all the outlier options *on*, and *off*. The results show that with all the outlier options on, BIRCH is not slower but faster, and at the same time, its quality is much better.

Figure 16. Time Scalability with respect to Increasing Number of Points per Cluster



Memory Size: In Phase 1, as memory size (or the maximum tree size) increases, (1) the running time increases because of processing a larger tree per rebuild, but only slightly because it is done in memory; (2) more but finer subclusters are generated to feed the next phase, and hence this results in better quality; (3) the inaccuracy caused by insufficient memory can be compensated to some extent by Phase 4 refinements. In other words, BIRCH can tradeoff memory versus time to achieve similar final quality.

6.7. Scalability

Three distinct ways of increasing the dataset size were used to test the scalability of BIRCH.

Increasing the Number of Points per Cluster (n): For each of DS1, DS2 and DS3, we created a range of datasets by keeping the generator settings the same except for changing n_l and n_h to change n, and hence N. Since N does not grow too far from that of the base workload, we decided to use the same amount of memory for these scaling experiments as we used for the base workload. This enables us to estimate for each pattern, given a fixed amount of memory, how large a dataset BIRCH can cluster while maintaining its stable qualities. Based on the performance analysis in 6.1, with M, P, d, K fixed, and only N growing, the running time should scale up linearly with N.

Table 7. Quality Stability with respect to Increasing Number of Points per Cluster

DS	n in $n \in [n_l..n_h]$	N	D/D_{int}
1	50..50	5000	1.87/1.99
	100..100	10000	1.92/2.01
	250..250	25000	1.87/1.99
	500..500	50000	1.86/1.98
	750..750	75000	1.88/2.00
	1000..1000	100000	1.87/2.00
	2500..2500	250000	1.88/2.00
2	50..50	5000	1.98/1.98
	100..100	10000	2.00/2.00
	250..250	25000	2.00/2.00
	500..500	50000	2.00/2.00
	750..750	75000	1.99/1.99
	1000..1000	100000	1.99/2.00
	2500..2500	250000	1.99/1.99
3	0..100	5000	4.08/4.42
	0..200	10000	4.30/4.78
	0..500	25000	4.21/4.65
	0..1000	50000	4.00/4.27
	0..1500	75000	3.74/4.22
	0..2000	100000	3.95/4.18
	0..5000	250000	4.23/4.52

Figure 17. Time Scalability with respect to Increasing Number of Clusters

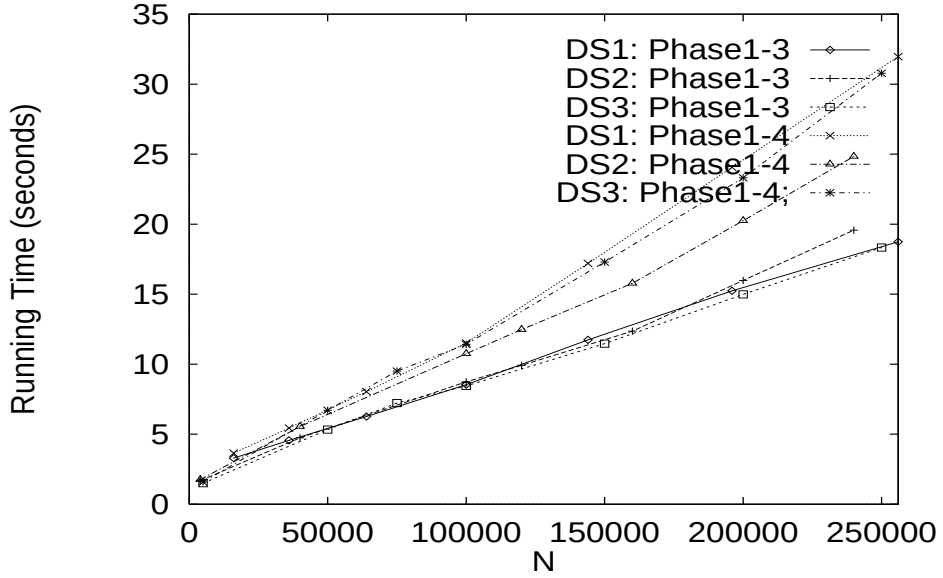


Table 8. Quality Stability with respect to Increasing Number of Clusters

DS	K	N	D/D_{int}
1	16	16000	2.32/2.00
	36	36000	1.98/2.00
	64	64000	1.93/1.99
	100	100000	1.87/2.00
	144	144000	1.87/1.99
	196	196000	1.87/2.00
	256	256000	1.87/2.00
2	4	4000	1.98/1.99
	40	40000	1.99/1.99
	100	100000	1.99/2.00
	120	120000	2.00/2.00
	160	160000	2.00/2.00
	200	200000	1.99/1.99
	240	240000	1.99/1.99
3	5	5000	5.57/6.43
	50	50000	4.10/4.52
	75	75000	4.04/4.76
	100	100000	3.95/4.18
	150	150000	4.21/4.26
	200	200000	5.22/4.49
	250	250000	5.52/4.48

Figure 18. Time Scalability with respect to Increasing Dimension

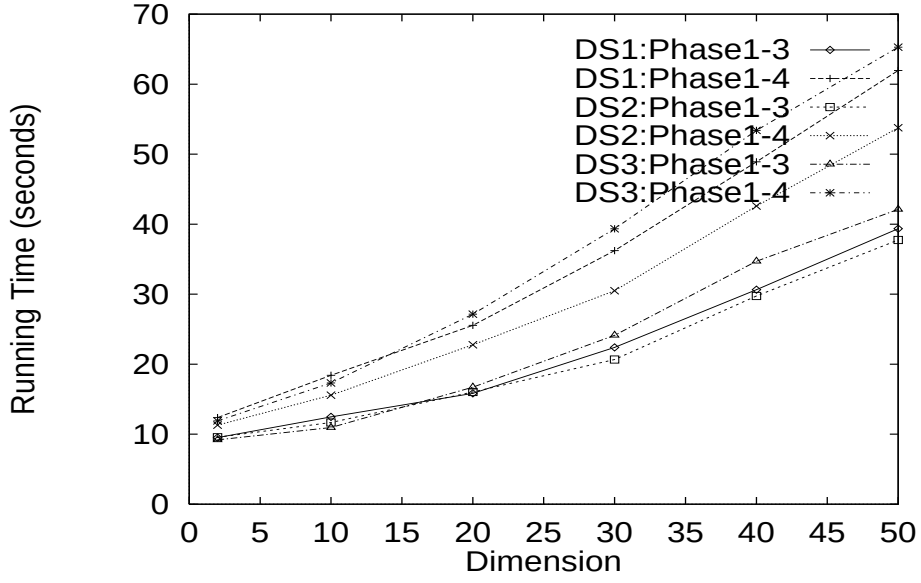


Table 9. Quality is Stable with respect to Increasing Dimension

DS	d (dimension)	$\bar{D}/\bar{D}_{int}$
1	2	1.87/1.99
	10	4.68/4.46
	20	6.52/6.31
	30	7.87/7.74
	40	8.97/8.93
	50	10.08/9.99
2	2	1.99/1.99
	10	4.46/4.46
	20	6.31/6.31
	30	7.74/7.74
	40	8.93/8.93
	50	10.02/9.99
3	2	3.95/4.28
	10	11.96/9.62
	20	17.11/13.62
	30	20.76/16.70
	40	25.34/19.28
	50	26.64/21.56

Following are the experimental results. With all three patterns of datasets, their running times for the first 3 phases, as well as for all 4 phases are plotted against the dataset size N in Figure 16. One can observe that for all three patterns of datasets: (1) The first 3 phases, as well as all 4 phases indeed scale up linearly with respect to N. (2) The running times for the first 3 phases grow similarly for all three patterns. (3) The improved 'nearest neighbor' algorithm used in Phase 4 is slightly sensitive to input data patterns. It works best for the sine pattern because there are usually less cluster centers **around** a data point in that pattern.

Table 7 provides the corresponding quality values of the intended clusters ($\bar{D}_{int}$) and of BIRCH clusters ($\bar{D}$) as n and N increase for all three patterns. It is shown from the table that with the same amount of memory, for a wide range of n and N, the quality of BIRCH clusters (indicated by $\bar{D}$) is consistently close to (or better than, due to the correction of 'outsiders') that of the intended clusters (indicated by $\bar{D}_{act}$).

Increasing the Number of Clusters (K): For each of DS1, DS2 and DS3, we create a range of datasets by keeping the generator settings the same except for changing K to N. Again, since K does not grow too far from that of the base workload, we decided to use the same amount of memory for these scaling experiments as we used for the base workload. The running time for the first 3 phases, as well as for all 4 phases are plotted against the dataset size N in Figure 17. (1) Again, the first 3 phases are confirmed to scale up linearly with respect to N for all three patterns. (2) The running times for all 4 phases are linear in N, but have slightly different slopes for the three different patterns of datasets. More specifically, for the grid pattern the slope is the largest, and for the sine pattern, the slope is the smallest. This is due to the fact that K and N are growing at the same time, and the complexity of Phase 4 is $O(K * N)$ (not linear to N) in the

worst case. Although we have tried to improve the Phase 4 refining algorithm using the 'nearest neighbor' techniques proposed in [14], and this improvement performs very well and brings the time complexity $O(N * K)$ down to be almost linear with respect to N, the linear slope is sensitive to the distribution patterns of data. In our case, for the grid pattern, since there are usually more cluster centers **around** a data point, it needs more time to find the nearest one; whereas for the sine pattern, since there are usually less cluster centers **around** a data point, it needs less time to find the nearest one; the random pattern is hence in the middle.

As for quality stability, Table 8 shows that with the same amount of memory, for a wide range of K and N, the quality of BIRCH clusters (indicated by $\bar{D}$) is again consistently close to (or better than) that of the intended clusters (indicated by $\bar{D}_{int}$).

Increasing the Dimension (d): For each of DS1, DS2 and DS3, we create a range of datasets by keeping the generator settings the same except for changing the dimension (d) from 2 to 50 to change the dataset size. In this experiment, the amount of memory used for each dataset is scaled up based on the dataset size (5% of the dataset size). With all three patterns of datasets, their running times for the first 3 phases, as well as for all 4 phases are plotted against the dimension in Figure 18. We observe that the running time curves deviate slightly from linear as the dimension and the corresponding memory increase. This is caused by the following fact: with M_0 (a constant amount of memory), the memory corresponding to a given d-dimensional dataset is scaled up as $M_0 * d$. Now, if N and P are constant, as the dimension increases the time complexity will scale up as $1 + \log_{\frac{P}{C*d}} \frac{M_0*d}{P}$ according to the analysis in Section 6.1. That is, as the dimension and memory increase, first the CF-tree size increases; second the branching factor decreases and causes the CF-tree height to increase. So for a larger d, incorporating a new data point goes through more levels on a larger CF-tree, and hence needs more time. The interesting thing is that by tuning P, one can make the scaling curves sublinear or superlinear, and in this case, with P=1024 bytes, the curves are slightly superlinear.

As for quality stability, Table 9 shows that for a wide range of d, the quality of BIRCH clusters (indicated by $\bar{D}$) are once again consistently close to (or better than) that of the intended clusters (indicated by $\bar{D}_{int}$).

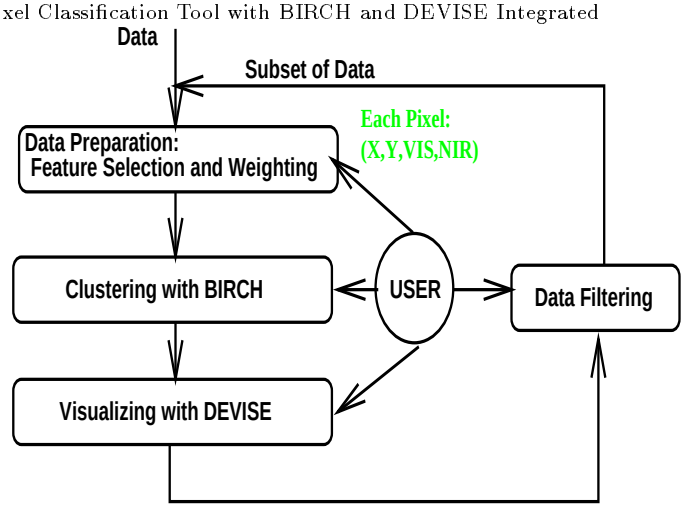
7. BIRCH Applications

In this section, we would like to show (1) how a clustering system like BIRCH can be used to help solve real-world problems, and (2) how BIRCH, CLARANS and KMEANS perform on some real datasets.

7.1. Interactive and Iterative Pixel Classification

The first application is motivated by the MVI (Multiband Vegetation Imager) technique developed in [17, 18]. The MVI is the combination of a charge-coupled device

Figure 19. Pixel Classification Tool with BIRCH and DEVISE Integrated



(CCD) camera, a filter exchange mechanism, and laptop computer used to capture rapid, successive images of plant canopies in two wavelength bands. One image is taken in the visible wavelength band, and the other in the near-infrared band. The purpose of using two wavelength bands is to allow for identification of different canopy components such as sunlit and shaded leaf area, sunlit and shaded branch area, clouds, and blue sky for studying plant canopy architecture. This is important to many fields including ecology, forestry, meteorology, and other agricultural sciences. The main use of BIRCH is to help classify pixels in the MVI images by performing clustering, and experimenting with different feature selection and weighting choices.

To do that, we integrated *BIRCH* with the *DEVISE* [3] data visualization system, as shown in Figure 19, to form a user-friendly, interactive and iterative pixel classification tool. As data is read in, (1) it is converted into the desired format; (2) interesting features are selected and weighted by the user interactively; (3) BIRCH is used for clustering the data in the space of the selected and weighted features; (4) relevant results such as the clusters as well as their corresponding data are visualized by *DEVISE* to enable the user to look for 'patterns' or to evaluate 'qualities'; (5) with feedback obtained from the visualizations, the user may choose to filter out a subset of the data which corresponds to some clusters, and/or re-adjust the feature selection and weighting for further clustering and visualization. The iteration with the above five major steps can be repeated, and the history of interaction can be maintained and manipulated as a tree structure which allows the user to decide what part of results to keep, where to proceed or to backtrack.

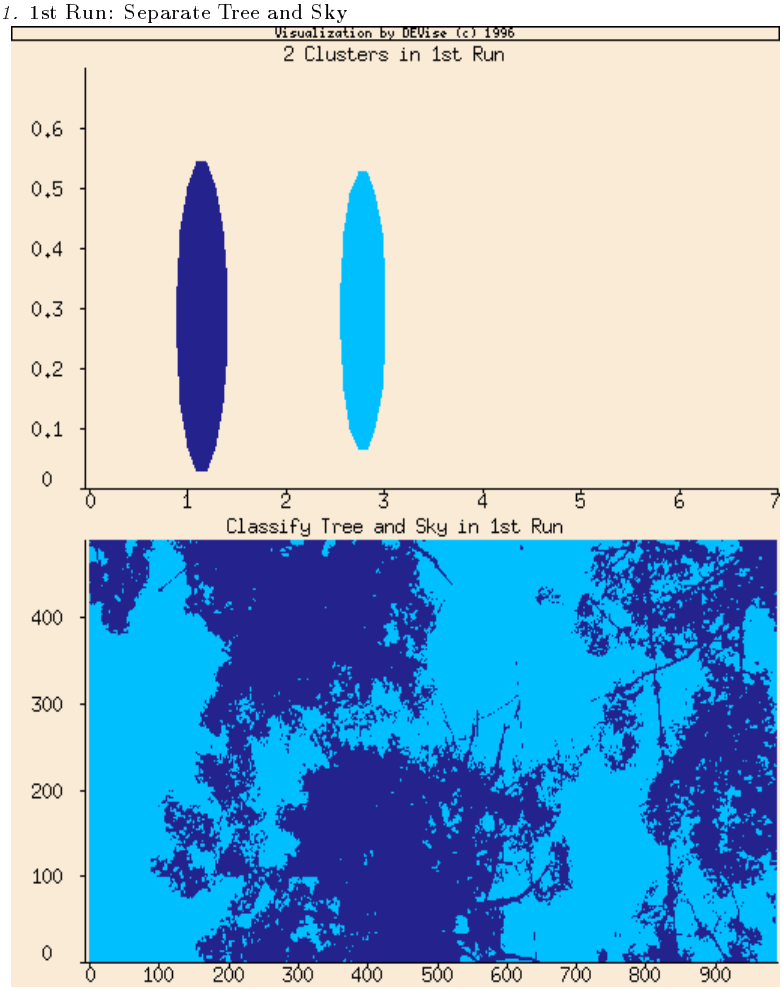
Following is an example of using this tool to help separate pixels in a MVI image. Figure 20 is a MVI image which contains two similar images of trees with the sky

Figure 20. The images taken in NIR and VIS

as background. The top picture is taken in the near-infrared band (NIR image), and the bottom one is taken in the visible wavelength band (VIS image). Each image contains 490 x 990 pixels after 'cutting' the noisy frames. Each pixel can be represented by a tuple with schema (x, y, nir, vis), where x and y are the coordinates of the pixel, and nir and vis are the corresponding brightness values in the NIR image and the VIS image respectively.

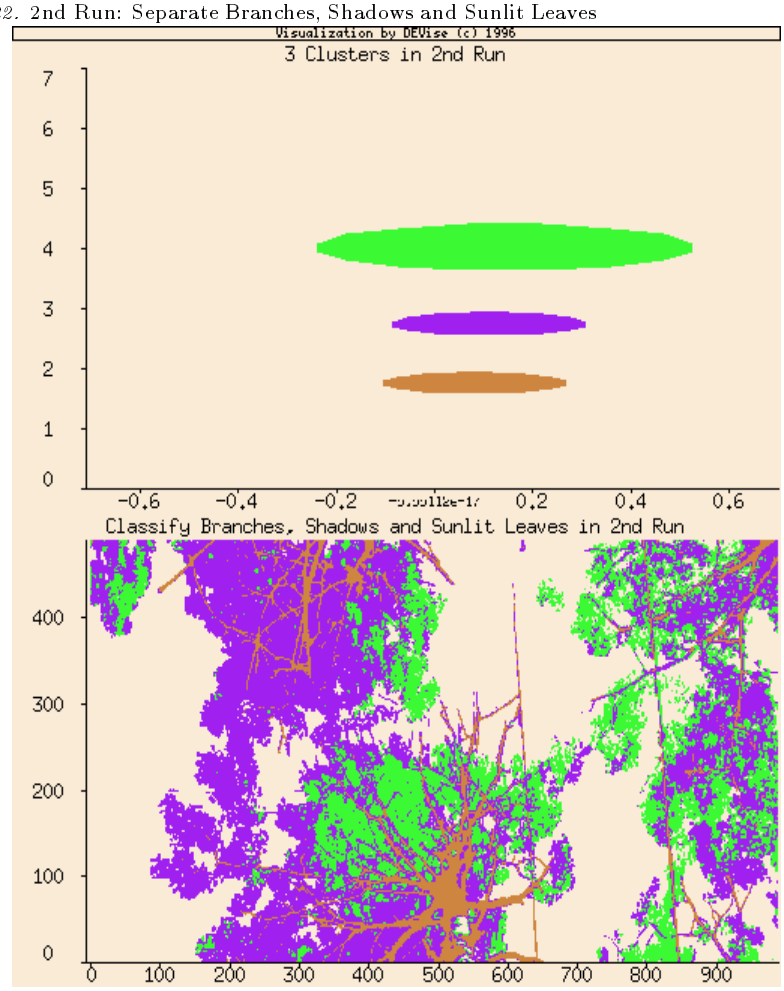
We start the first iteration with the number of clusters set to 2, in the hope of finding two clusters corresponding to the trees and the sky. It is easy to notice that the trees and the sky are better differentiated in the VIS image than in the NIR image. So the weight assigned to vis is 10 times more than the weight assigned

Figure 21. 1st Run: Separate Tree and Sky



to *nir*. Then *BIRCH* is invoked under the default settings to do the clustering which takes a total of 50 seconds (including two scans of the VIS and NIR values from an ASCII file on disk in Phase 1 and Phase 4; each scan takes about 4.5 seconds). Figure 21 is the *DEVISE* visualization of the clusters obtained after the first iteration (top: where x-axis is for weighted *vis*, y-axis is for weighted *nir*, each cluster is plotted as a circle with the centroid as the center and the standard deviation as the radius) as well as the corresponding parts of the image (bottom: where x-axis is for the x coordinates of pixels and y-axis is for the y coordinates of pixels).

Figure 22. 2nd Run: Separate Branches, Shadows and Sunlit Leaves



Visually, by comparison with the original images, one can see that the two clusters form a satisfactory classification of the trees and the sky. In general, it may take a few iterations for the user to identify a good set of weights that achieves such a classification. However, the important point is that once a set of good weights are found from one image pair, they can be used for classifying several other image pairs taken under the same conditions (the same kind of tree, the same weather conditions, about the same time in a day), and the pixel classification task becomes automatic, without further human intervention.

In the second iteration, the part of the data that corresponds to the trees is filtered out for further clustering. The number of clusters is set at 3 in the hope of

Table 10. BIRCH,CLARANS,KMEANS on Pixel Classification

Method	Image1: N=485100,K=2			Image2: N=485100,K=5		
	Time	$\bar{D}$	#Scan	Time	$\bar{D}$	#Scan
BIRCH	50	0.5085	2	53	0.6269	2
CLARANS	368	0.5061	330	642	0.6292	523
KMEANS	8.3	0.5015	5	30	0.6090	48

finding three clusters which correspond to branches, shadows and sunlit leaves. One can observe from the original images that the branches, shadows and sunlit leaves are easier to tell apart from the NIR image than from the VIS image. So we now weight *nir* 10 times heavier than *vis*. This time, with the same amount of memory, but a smaller set of data (263401 tuples versus 485100 tuples), the clustering can be done at a much finer granularity, which should result in better quality. It takes BIRCH a total of 29.08 seconds (including two scans of the subset of VIS and NIR values from an ASCII file on disk). Figure 22 shows the clusters after the second iteration as well as the corresponding parts of the image.

We used two very different MVI image pairs to compare how BIRCH, CLARANS and KMEANS perform relative to each other. Table 10 summarizes the results. For BIRCH, the memory used is only 5% of the dataset size, whereas for CLARANS and KMEANS, the whole dataset as well as related secondary data structures are all in memory. For both cases, BIRCH, CLARANS and KMEANS have almost the same quality. The quality obtained by CLARANS and KMEANS is slightly better than that obtained by BIRCH. To explain this, one should notice that (1) CLARANS and KMEANS are doing 'hill-climbing' and they stop only after they reach some 'optimal' clustering; (2) in this application, N is not too big, K and d are extremely small, and the pixel distribution in terms of VIS and NIR values is very simple in modality, so the 'hill' that they climb tends to be simple too. Very bad local optimal solutions do not prevail, and CLARANS and KMEANS can usually reach a pretty good clustering in the 'hill'. BIRCH stops after scanning the dataset only twice, and it reaches a clustering almost as good as those optimal ones found by CLARANS and KMEANS with a lot of additional data scans. BIRCH's running time is not sensitive to the different MVI image pairs whereas for CLARANS and KMEANS, their running time and data scan numbers are very sensitive to the datasets themselves.

7.2. Codebook Generalization in Image Compression

Digital image compression [19] is the technology of reducing image data to save storage space and transmission bandwidth. *Vector quantization* [11] is a widely used image compression/decompression technique which operates on blocks of pixels instead of pixels for better efficiency. In vector quantization, the original image is first decomposed into small rectangular blocks, and each block is represented as a vector. Given a codebook of size K, it contains K codewords that are vectors

Table 11. BIRCH, CLARANS and LBG on Image Compression

Method	Lena				Baboon			
	Time	#Scans	Distortion	Entropy	Time	#Scan	Distortion	Entropy
BIRCH	15	9	712	6.70	17	8	5097	7.15
CLARANS	2222	8146	693	7.43	2111	7651	5070	7.70
LBG	19	46	719	7.38	31	42	5155	7.73

serving as seeds to attract other vectors based upon the nearest neighbor criterion. Each vector is **encoded** with the *codebook*, i.e., finding its *nearest codeword* from the *codebook*, and later is **decoded** with the same *codebook*, i.e., using its *nearest codeword* in the *codebook* as its value.

Given the training vectors (from the training image) and the desired codebook size (i.e., number of codewords), the main problem of vector quantization is how to generate the codebook. A commonly used codebook generating algorithm is the LBG algorithm [20]. LBG is just KMEANS except for two specific modifications that must be made for use in image compression: (1) To avoid getting stuck at a bad local optimal LBG starts with an initial codebook of size 1 (instead of the desired size), and proceeds by refining and splitting the codebook iteratively. (2) If empty cells (i.e., codewords that attract no vectors) in the codebook are found during the refinement, it has a strategy of filling the empty cells via splitting.

In a little more detail:

1. LBG uses the *GLA* (or KMEANS with empty cell filling strategy) algorithm [11] to find the 'optimal' codebook of current size.

2. If it reaches the desired codebook size, then it stops; otherwise it doubles the current codebook size, perturbs and splits the current 'optimal' codebook and goes to the previous step.

The above *LBG* algorithm invokes many extra scans of the training vectors during the codebook optimizations at all the codebook size levels before reaching the desired codebook size level; yet, of course, it can not completely escape from locally optimal solutions.

With BIRCH clustering the training vectors, we know that from the first 3 phases, with a single scan of the training vectors, the clusters obtained generally capture the major vector distribution patterns and only have minor inaccuracies. So in Phase 3, if we set the number of clusters directly as the desired codebook size, and use the centroids of the obtained clusters as the initial codebook, then we can feed them to *GLA* for further optimization. So, in contrast with *LBG*, (1) the initial codebook from the first 3 phases of BIRCH is not likely to lead to a bad locally optimal codebook; (2) using BIRCH to generate the codebook will involve fewer scans of the training vectors.

We have used two different images: *Lena* and *Baboon* (each with 512x512 pixels) as examples to compare BIRCH, CLARANS and LBG's performances in terms of the running time, number of scans of the dataset, distortion and entropy [19, 26]

on high dimensional (d=16) real datasets. First the training vectors are derived by blocking the image into 4x4 blocks (d=16), and the desired codebook size is set to 256. Distortion is defined as the sum of squares of Euclidean distances from all training vectors to their nearest codewords. It is a widely used quality measurement, and smaller distortion values imply better qualities. Entropy is the average number of bits needed for encoding a training vector, so lower entropy means better compression.

Table 11 summarizes the performance results. *Time* is the total time to generate the initial codebook of the desired size (256) and then use GLA algorithm to refine the codebook. *#Scans* denotes the total number of scans of the dataset in order to reach the final codebook. *Distortion* and *Entropy* are obtained by compressing the images using the final codebook. One can see that for both images, (1) for all four aspects listed in the table, using BIRCH to generate the initial codebook is consistently better than using CLARANS or using LBG; (2) for running time, data scans and entropy, using BIRCH is significantly better than using CLARANS. For CLARANS, its running time is much longer than that of BIRCH, whereas its final codebook is only slightly better than that obtained through BIRCH (and is better only in terms of distortion). Readers can look at the compressed images in Figure 23 through 28 for easy visual quality comparisons.

8. Summary and Future Research

BIRCH provides a clustering method for very large datasets. It makes a large clustering problem tractable by concentrating on densely occupied portions, and creating a compact summary. It utilizes measurements that capture the natural closeness of data and can be stored and updated incrementally in a height-balanced tree. BIRCH can work with any given amount of memory, and the I/O complexity is a little more than one scan of data. Experimentally, BIRCH is shown to perform very well on several large datasets, and is significantly superior to CLARANS and KMEANS in terms of quality, speed, stability and scalability overall.

Proper parameter setting and further generalization are two important topics to explore in the future. Additional issues include other heuristic methods for increasing the threshold dynamically, how to handle non-metric attributes, other threshold requirements and related insertion, rebuilding algorithms, and clustering confidence measurements. An important direction for further study is how to make use of the clustering information obtained from BIRCH to help solve problems such as storage optimization, data partition and index construction.

Notes

1. The reducibility property requires that if for clusters i, j, k, and some distance value θ: $d(i,j) < \theta$, $d(i,k) > \theta$, $d(j,k) > \theta$, then for the merged cluster $i+j$, $d(i+j,k) > \theta$.

References

1. Norbert. Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger, *The R*-tree: An Efficient and Robust Access Method for Points and Rectangles*, Proc. of ACM SIGMOD Int. Conf. on Management of Data, 322-331,1990.
2. Peter Cheeseman, James Kelly, Matthew Self, et al., *AutoClass : A Bayesian Classification System*, Proc. of the 5th Int. Conf. on Machine Learning, Morgan Kaufman, Jun. 1988.
3. Michael Cheng, Miron Livny, and Raghu Ramakrishnan, *Visual Analysis of Stream Data*, Proc. of IS&T/SPIE Conf. on Visual Data Exploration and Analysis, San Jose, CA, Feb. 1995.
4. Richard Duda, and Peter E. Hart, *Pattern Classification and Scene Analysis*, Wiley, 1973.
5. R. Dubes, and A.K. Jain, *Clustering Methodologies in Exploratory Data Analysis*, Advances in Computers, Edited by M.C. Yovits, Vol. 19, Academic Press, New York, 1980.
6. Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu, *A Database Interface for Clustering in Large Spatial Databases*, Proc. of 1st Int. Conf. on Knowledge Discovery and Data Mining, 1995.
7. Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu, *Knowledge Discovery in Large Spatial Databases: Focusing Techniques for Efficient Class Identification*, Proc. of 4th Int. Symposium on Large Spatial Databases, Portland, Maine, U.S.A., 1995.
8. E. A. Feigenbaum, and H. Simon, *EPAM-like models of recognition and learning*, Cognitive Science, vol. 8, 1984, 305-336.
9. Douglas H. Fisher, *Knowledge Acquisition via Incremental Conceptual Clustering*, Machine Learning, 2(2), 1987
10. Douglas H. Fisher, *Iterative Optimization and Simplification of Hierarchical Clusterings*, Technical Report CS-95-01, Dept. of Computer Science, Vanderbilt University, Nashville, TN 37235.
11. A. Gersho, and R. Gray, *Vector quantization and signal compression*, Boston, Ma.: Kluwer Academic Publishers, 1992.
12. John H. Gennari, Pat Langley, and Douglas Fisher, *Models of Incremental Concept Formation*, Artificial Intelligence, vol. 40, 1989, 11-61.
13. A. Guttman, *R-trees: a dynamic index structure for spatial searching*, Proc. ACM SIGMOD Int. Conf. on Management of Data, 47-57, 1984.
14. C. Huang, Q. Bi, G. Stiles, R. Harris, *Fast Full Search Equivalent Encoding Algorithms for Image Compression Using Vector Quantization*, IEEE Trans. on Image Processing, vol. 1, no. 3, July, 1992.
15. J. A. Hartigan, and M. A. Wong, *A K-Means Clustering Algorithm*, Appl. Statist., vol. 28, no. 1, 1979.
16. Leonard Kaufman, and Peter J. Rousseeuw, *Finding Groups in Data - An Introduction to Cluster Analysis*, Wiley Series in Probability and Mathematical Statistics, 1990.
17. C.J. Kucharik, and J.M. Norman, *Measuring Canopy Architecture with a Multiband Vegetation Imager (MVI)* Proc. of the 22nd conf. on Agricultural and Forest Meteorology, American Meteorological Society annual meeting, Atlanta, GA, Jan 28-Feb 2, 1996.
18. C.J. Kucharik, J.M. Norman, L.M. Murdock, and S.T. Gower, *Characterizing Canopy nonrandomness with a Multiband Vegetation Imager (MVI)*, Submitted to Journal of Geophysical Research, to appear in the Boreal Ecosystem-Atmosphere Study (BOREAS) special issue.
19. Weidong Kou, *Digital Image Compression Algorithms and Standards*, Kluwer Academic Publishers, 1995.
20. Y. Linde, A. Buzo, and R. M. Gray, *An Algorithm for Vector Quantization Design*, IEEE Trans. on Communications, vol. 28, no. 1, 1980.
21. Michael Lebowitz, *Experiments with Incremental Concept Formation : UNIMEM*, Machine Learning, 1987.
22. R.C.T.Lee, *Clustering analysis and its applications*, Advances in Information Systems Science, Edited by J.T.Toum, Vol. 8, pp. 169-292, Plenum Press, New York, 1981.
23. F. Murtagh, *A Survey of Recent Advances in Hierarchical Clustering Algorithms*, The Computer Journal, 1983.

24. Raymond T. Ng and Jiawei Han, *Efficient and Effective Clustering Methods for Spatial Data Mining*, Proc. of VLDB, 1994.
25. Clark F. Olson, *Parallel Algorithms for Hierarchical Clustering*, Technical Report, Computer Science Division, Univ. of California at Berkeley, Dec.,1993.
26. Majid Rabbani, and Paul W. Jones, *Digital Image Compression Techniques*, SPIE Optical Engineering Press, 1991.
27. Tian Zhang, Raghu Ramakrishnan, and Miron Livny, *BIRCH: An Efficient Data Clustering Method for Very Large Databases*, Technical Report, Computer Sciences Dept., Univ. of Wisconsin-Madison, 1995.
28. Tian Zhang, *Data Clustering for Very Large Datasets Plus Applications*, Dissertation, Computer Sciences Dept. at Univ. of Wisconsin-Madison, 1996.

Received Date
Accepted Date
Final Manuscript Date

Figure 23. Lena Compressed with BIRCH Codebook

Figure 24. Lena Compressed with CLARANS Codebook

Figure 25. Lena Compressed with LBG Codebook

Figure 26. Baboon Compressed with BIRCH Codebook

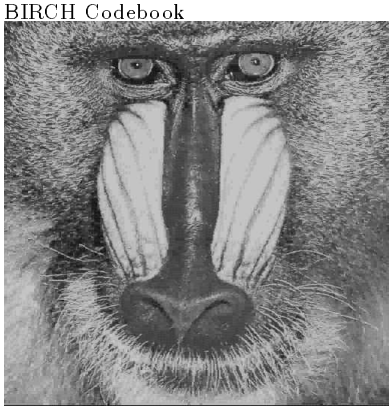


Figure 27. Baboon Compressed with CLARANS Codebook

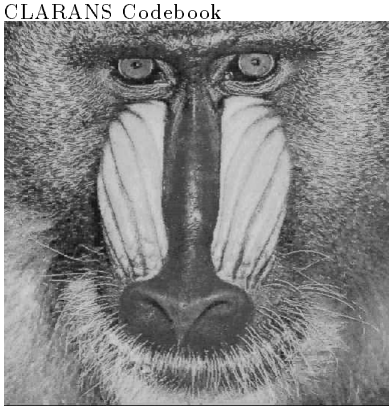


Figure 28. Baboon Compressed with LBG Codebook