CURE: An Efficient Clustering Algorithm for Large Databases

Sudipto Guha*
Stanford University
Stanford, CA 94305
sudipto@cs.stanford.edu

Rajeev Rastogi
Bell Laboratories
Murray Hill, NJ 07974
rastogi@bell-labs.com

Kyuseok Shim
Bell Laboratories
Murray Hill, NJ 07974
shim@bell-labs.com

Abstract

Clustering, in data mining, is useful for discovering groups and identifying interesting distributions in the underlying data. Traditional clustering algorithms either favor clusters with spherical shapes and similar sizes, or are very fragile in the presence of outliers. We propose a new clustering algorithm called CURE that is more robust to outliers, and identifies clusters having non-spherical shapes and wide variances in size. CURE achieves this by representing each cluster by a certain fixed number of points that are generated by selecting *well scattered* points from the cluster and then shrinking them toward the center of the cluster by a specified fraction. Having more than one representative point per cluster allows CURE to adjust well to the geometry of non-spherical shapes and the shrinking helps to dampen the effects of outliers. To handle large databases, CURE employs a combination of *random sampling* and *partitioning*. A random sample drawn from the data set is first partitioned and each partition is *partially* clustered. The partial clusters are then clustered in a second pass to yield the desired clusters. Our experimental results confirm that the quality of clusters produced by CURE is much better than those found by existing algorithms. Furthermore, they demonstrate that random sampling and partitioning enable CURE to not only outperform existing algorithms but also to scale well for large databases without sacrificing clustering quality.

1 Introduction

The wealth of information embedded in huge databases belonging to corporations (e.g., retail, financial, telecom) has spurred a tremendous interest in the areas of *knowledge discovery* and *data mining*. Clustering, in data mining, is a useful technique for discovering interesting data distributions and patterns in the underlying data. The problem of clustering can be defined as follows: given n data points in a d-dimensional metric space, partition the data points into k clusters such that the data points within a cluster are more similar to each other than data points in different clusters.

*The work was done while the author was visiting Bell Laboratories.

1.1 Traditional Clustering Algorithms - Drawbacks

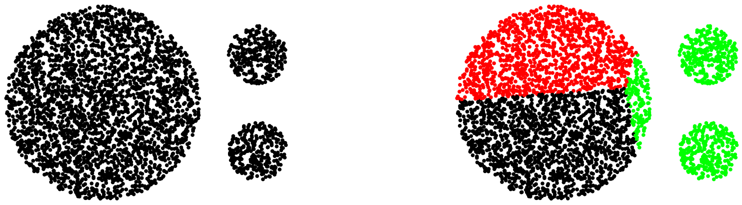
Existing clustering algorithms can be broadly classified into *partitional* and *hierarchical* [JD88]. Partitional clustering algorithms attempt to determine k partitions that optimize a certain criterion function. The square-error criterion, defined below, is the most commonly used (m_i is the mean of cluster C_i).

$$E = \sum_{i=1}^{k} \sum_{p \in C_i} \|p - m_i\|^2.$$

The square-error is a good measure of the within-cluster variation across all the partitions. The objective is to find k partitions that minimize the square-error. Thus, square-error clustering tries to make the k clusters as compact and separated as possible, and works well when clusters are compact clouds that are rather well separated from one another. However, when there are large differences in the sizes or geometries of different clusters, as illustrated in Figure 1, the square-error method could split large clusters to minimize the square-error. In the figure, the square-error is larger for the three separate clusters in (a) than for the three clusters in (b) where the big cluster is split into three portions, one of which is merged with the two smaller clusters. The reduction in square-error for (b) is due to the fact that the slight reduction in square error due to splitting the large cluster is weighted by many data points in the large cluster.

A hierarchical clustering is a sequence of partitions in which each partition is nested into the next partition in the sequence. An *agglomerative* algorithm for hierarchical clustering starts with the disjoint set of clusters, which places each input data point in an individual cluster. Pairs of items or clusters are then successively merged until the number of clusters reduces to k. At each step, the pair of clusters merged are the ones between which the distance is the minimum. The widely used measures for distance between clusters are as follows (m_i is the mean for cluster C_i and n_i is the number of points in C_i).

$$\begin{aligned}
d_{mean}(C_i, C_j) &= \|m_i - m_j\| \\
d_{ave}(C_i, C_j) &= 1/(n_i n_j) \sum_{p \in C_i} \sum_{p' \in C_j} \|p - p'\| \\
d_{max}(C_i, C_j) &= \max_{p \in C_i, p' \in C_j} \|p - p'\|
\end{aligned}$$



(a) (b)

Figure 1: Splitting of a large cluster by partitional algorithms

$$d_{min}(C_i, C_j) \quad = \quad \min_{p \in C_i, p' \in C_j} \|p - p^{'}\|$$

For example, with d_{mean} as the distance measure, at each step, the pair of clusters whose centroids or means are the closest are merged. On the other hand, with d_{min}, the pair of clusters merged are the ones containing the closest pair of points. All of the above distance measures have a minimum variance flavor and they usually yield the same results if the clusters are compact and well-separated. However, if the clusters are close to one another (even by outliers), or if their shapes and sizes are not hyperspherical and uniform, the results of clustering can vary quite dramatically. For example, with the data set shown in Figure 1(a), using d_{max}, d_{ave} or d_{mean} as the distance measure results in clusters that are similar to those obtained by the square-error method shown in Figure 1(b). Similarly, consider the example data points in Figure 2. The desired elongated clusters are shown in Figure 2(a). However, d_{mean} as the distance measure, causes the elongated clusters to be split and portions belonging to neighboring elongated clusters to be merged. The resulting clusters are as shown in Figure 2(b). On the other hand, with d_{min} as the distance measure, the resulting clusters are as shown in Figure 2(c). The two elongated clusters that are connected by narrow string of points are merged into a single cluster. This "chaining effect" is a drawback of d_{min} – basically, a few points located so as to form a bridge between the two clusters causes points across the clusters to be grouped into a single elongated cluster.

From the above discussion, it follows that neither the centroid-based approach (that uses d_{mean}) nor the all-points approach (based on d_{min}) work well for non-spherical or arbitrary shaped clusters. A shortcoming of the centroid-based approach is that it considers only one point as representative of a cluster – the cluster centroid. For a large or arbitrary shaped cluster, the centroids of its subclusters can be reasonably far apart, thus causing the cluster to be split. The all-points approach, on the other hand, considers all the points within a cluster as representative of the cluster. This other extreme, has its own drawbacks, since it makes the clustering algorithm extremely sensitive to outliers and to slight changes in the position of data points.

When the number N of input data points is large, hierarchical clustering algorithms break down due to their non-linear time complexity (typically, $O(N^2)$) and huge I/O costs. In order to remedy this problem, in [ZRL96], the authors propose a new clustering method named BIRCH, which represents the state of the art for clustering large data sets. BIRCH first performs a *preclustering phase* in which dense regions of points are represented by compact summaries, and then a centroid-based hierarchical algorithm is used to cluster the set of summaries (which is much smaller than the original dataset).

The preclustering algorithm employed by BIRCH to reduce input size is incremental and approximate. During preclustering, the entire database is scanned, and cluster summaries are stored in memory in a data structure called the CF-tree. For each successive data point, the CF-tree is traversed to find the closest cluster to it in the tree, and if the point is within a threshold distance of the closest cluster, it is absorbed into it. Otherwise, it starts its own cluster in the CF-tree.

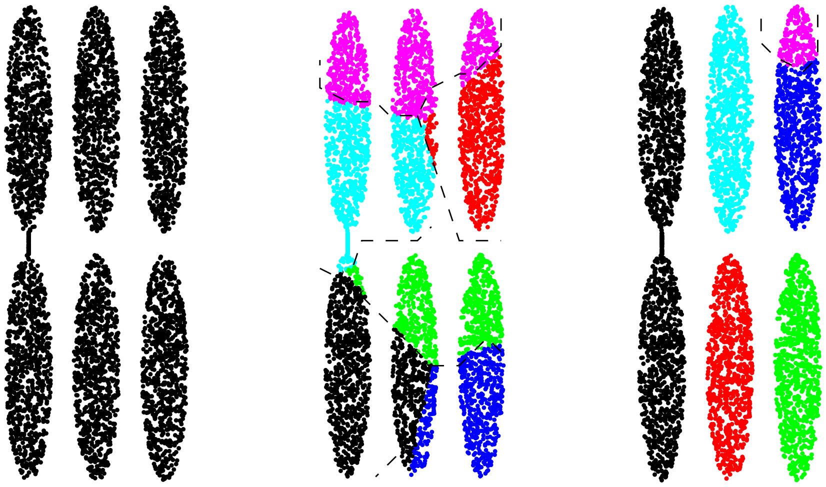
Once the clusters are generated, a final labeling phase is carried out in which using the centroids of clusters as seeds, each data point is assigned to the cluster with the closest seed. Using only the centroid of a cluster when redistributing the data in the final phase has problems when clusters do not have uniform sizes and shapes as in Figure 3(a). In this case, as illustrated in Figure 3(b), in the final labeling phase, a number of points in the bigger cluster are labeled as belonging to the smaller cluster since they are closer to the centroid of the smaller cluster.

1.2 Our Contributions

In this paper, we propose a new clustering method named CURE (Clustering Using Representatives) whose salient features are described below.

Hierarchical Clustering Algorithm: CURE employs a novel hierarchical clustering algorithm that adopts a middle ground between the centroid-based and the all-point extremes. In CURE, a constant number c of *well scattered* points in a cluster are first chosen. The scattered points capture the shape and extent of the cluster. The chosen scattered points are next shrunk towards the centroid of the cluster by a fraction α. These scattered points after shrinking are used as representatives of the cluster. The clusters with the closest pair of representative points are the clusters that are merged at each step of CURE's hierarchical clustering algorithm.

The scattered points approach employed by CURE alleviates the shortcomings of both the all-points as well as the centroid-based approaches. It enables CURE to correctly identify the clusters in Figure 2(a) – the resulting clusters due to the centroid-based and all-points approaches is as shown in Figures 2(b) and 2(c), respectively. CURE is less sensitive to outliers since shrinking the scattered points toward the mean dampens the adverse effects due to outliers – outliers are typically further away from the mean and are thus shifted a larger distance due to the shrinking. Multiple scattered points also enable CURE to discover non-spherical clusters like the elongated clusters shown in Figure 2(a). For the centroid-based algorithm, the space that constitutes the vicinity of the single centroid for a cluster is spherical. Thus, it favors spherical clusters and as shown in Figure 2(b), splits the elongated clusters. On the other hand, with multiple scattered points as representatives of a cluster, the space



(a) (b) (c)

Figure 2: Clusters generated by hierarchical algorithms

(a) (b)

Figure 3: Problem of labeling

that forms the vicinity of the cluster can be non-spherical, and this enables CURE to correctly identify the clusters in Figure 2(a).

Note that the kinds of clusters identified by CURE can be tuned by varying α between 0 and 1. CURE reduces to the centroid-based algorithm if $\alpha = 1$, while for $\alpha = 0$, it becomes similar to the all-points approach. CURE's hierarchical clustering algorithm uses space that is linear in the input size n and has a worst-case time complexity of $O(n^2 \log n)$. For lower dimensions (e.g., two), the complexity can be shown to further reduce to $O(n^2)$. Thus, the time complexity of CURE is no worse than that of the centroid-based hierarchical algorithm.

Random Sampling and Partitioning: CURE's approach to the clustering problem for large data sets differs from BIRCH in two ways. First, instead of preclustering with all the data points, CURE begins by drawing a random sample from the database. We show, both analytically and experimentally, that random samples of moderate sizes preserve information about the geometry of clusters fairly accurately, thus enabling CURE to correctly cluster the input. In particular, assuming that each cluster has a certain minimum size, we use chernoff bounds to calculate the minimum sample size for which the sample contains, with high probability, at least a fraction f of every cluster. Second, in order to further speed up clustering, CURE first partitions the random sample and partially clusters the data points in each partition. After eliminating outliers, the preclustered data in each partition is then clustered in a final pass to generate the final clusters.

Labeling Data on Disk: Once clustering of the random sample is completed, instead of a single centroid, multiple representative points from each cluster are used to label the remainder of the data set. The problems with BIRCH's labeling phase are eliminated by assigning each data point to the cluster containing the closest representative point.

Overview: The steps involved in clustering using CURE are described in Figure 4. Our experimental results confirm that not only does CURE's novel approach to clustering based on scattered points, random sampling and partitioning enable it to find clusters that traditional clustering algorithms fail to find, but it also results in significantly better execution times.

The remainder of the paper is organized as follows. In Section 2, we survey related work on clustering large data sets. We present CURE's hierarchical clustering algorithm that uses representative points, in Section 3. In Section 4, we discuss issues related to sampling, partitioning, outlier handling and labeling in CURE. Finally, in Section 5, we present the results of our experiments which support our claims about CURE's clustering ability and execution times. Concluding remarks are made in Section 6.

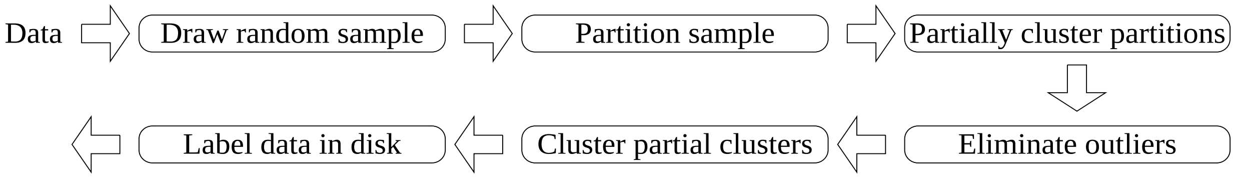


Figure 4: Overview of CURE

2 Related Work

In recent years, a number of clustering algorithms for large databases have been proposed [NH94, ZRL96, EKSX96]. In [NH94], the authors propose a partitional clustering method for large databases which is based on randomized search. Each cluster is represented by its *medoid*, the most centrally located point in the cluster, and the objective is to find the k best medoids that optimize the criterion function. The authors reduce this problem to that of graph search by representing each set of k medoids as a node in the graph, two nodes being adjacent if they have $k-1$ medoids in common. Initially, an arbitrary node is set to be the current node and a fixed number of iterations are performed. In each iteration, a random neighbor of the current node is set to be the current node if it results in better clustering. The computation of the criterion function for the random neighbor requires the entire database to be examined. It is experimentally shown that CLARANS outperforms the traditional *k-medoid* algorithms. However, CLARANS may require several passes over the database, the runtime cost of which could be prohibitive for large databases. Furthermore, like other partitional clustering algorithms, it could converge to a local optimum.

In [EKX95], the authors use the R*-tree[SRF87, BKSS90, Sam89] to improve the I/O efficiency of CLARANS on large databases by (1) drawing samples from leaf pages to reduce the number of data points (since data points are packed in leaf nodes based on spatial locality, a sample point in the leaf page can be a good representative point), and (2) focusing on relevant points when evaluating the "goodness" of a neighbor.

Since multiple I/O scans of the data points is a bottleneck for existing clustering algorithms, in [ZRL96], the authors present a clustering method named BIRCH whose I/O complexity is a little more than one scan of the data. BIRCH first pre-clusters the data into the maximum possible and finest possible subclusters that can fit in main-memory. For the pre-clustering phase, BIRCH employs a CF-tree which is a balanced tree structure similar to the B-tree and R-tree family[Sam89]. After pre-clustering, BIRCH treats each of the subcluster summaries as representative points, and runs a well-known approximation algorithm from [Ols93], which is an agglomerative hierarchical clustering algorithm.

BIRCH and CLARANS work well for convex or spherical clusters of uniform size. However, they are unsuitable when clusters have different sizes (see Figure 1), or when clusters are non-spherical (see Figure 2). For clustering such arbitrary shaped collections of points (e.g., ellipsoid, spiral, cylindrical), a density-based algorithm called DBSCAN was proposed in [EKSX96]. DBSCAN requires the user to specify two parameters that are used to define minimum density for clustering – the radius *Eps* of the neighborhood of a point and the minimum number of points *MinPts* in the neighborhood. Clusters are then found by starting from an arbitrary point and if its neighborhood satisfies the minimum density, including the points in its neighborhood into the cluster. The process is then repeated for the newly added points.

While DBSCAN can find clusters with arbitrary shapes, it suffers from a number of problems. DBSCAN is very sensitive to the parameters *Eps* and *MinPts*, which in turn, are difficult to determine. Furthermore, DBSCAN also suffers from the robustness problems that plague the all-points hierarchical clustering algorithm – in case there is a dense string of points connecting two clusters, DBSCAN could end up merging the two clusters. Also, DBSCAN does not perform any sort of preclustering and executes directly on the entire database. As a result, for large databases, DBSCAN could incur substantial I/O costs. Finally, with density-based algorithms, using random sampling to reduce the input size may not be feasible – the reason for this is that unless sample sizes are large, there could be substantial variations in the density of points within each cluster in the random sample.

3 Hierarchical Clustering Algorithm

In this section, we present CURE's hierarchical clustering algorithm whose salient features are: (1) the clustering algorithm can recognize arbitrarily shaped clusters (e.g., ellipsoidal), (2) the algorithm is robust to the presence of outliers, and (3) the algorithm has linear storage requirements and time complexity of $O(n^2)$ for low-dimensional data. The n data points input to the algorithm are either a sample drawn randomly from the original data points, or a subset of it if partitioning is employed. An analysis of issues related to the size of the random sample and number of partitions is presented in Section 4.

3.1 Intuition and Overview

The clustering algorithm starts with each input point as a separate cluster, and at each successive step merges the closest pair of clusters. In order to compute the distance between a pair of clusters, for each cluster, c representative points are stored. These are determined by first choosing c *well scattered* points within the cluster, and then shrinking them toward the mean of the cluster by a fraction α. The distance between two clusters is then the distance between the closest pair of representative points – one belonging to each of the two clusters. Thus, only the representative points of a cluster are used to compute its distance from other clusters.

The c representative points attempt to capture the physical shape and geometry of the cluster. Furthermore, shrinking the scattered points toward the mean by a factor α gets rid of surface abnormalities and mitigates the effects of outliers. The reason for this is that outliers typically will be further away from the cluster center, and as a result, the shrinking would cause outliers to move more toward the center while the remaining representative points would experience minimal shifts. The larger movements in the outliers would thus reduce their ability to cause the wrong clusters to

```
procedure cluster(S, k)
begin
1.  T := build_kd_tree(S)
2.  Q := build_heap(S)
3.  while size(Q) > k do {
4.     u := extract_min(Q)
5.     v := u.closest
6.     delete(Q, v)
7.     w := merge(u, v)
8.     delete_rep(T, u); delete_rep(T, v); insert_rep(T, w)
9.     w.closest := x /* x is an arbitrary cluster in Q */
10.    for each x ∈ Q do {
11.       if dist(w, x) < dist(w, w.closest)
12.          w.closest := x
13.       if x.closest is either u or v {
14.          if dist(x, x.closest) < dist(x, w)
15.             x.closest := closest_cluster(T, x, dist(x, w))
16.          else
17.             x.closest := w
18.          relocate(Q, x)
19.       }
20.       else if dist(x, x.closest) > dist(x, w) {
21.          x.closest := w
22.          relocate(Q, x)
23.       }
24.    }
25.    insert(Q, w)
26. }
end
```

Figure 5: Clustering algorithm

be merged. The parameter α can also be used to control the shapes of clusters. A smaller value of α shrinks the scattered points very little and thus favors elongated clusters. On the other hand, with larger values of α, the scattered points get located closer to the mean, and clusters tend to be more compact.

3.2 Clustering Algorithm

In this subsection, we describe the details of our clustering algorithm (see Figure 5). The input parameters to our algorithm are the input data set S containing n points in d-dimensional space and the desired number of clusters k. As we mentioned earlier, starting with the individual points as individual clusters, at each step the closest pair of clusters is merged to form a new cluster. The process is repeated until there are only k remaining clusters.

Data Structures: With every cluster is stored all the points in the cluster. Also, for each cluster u, u.mean and u.rep store the mean of the points in the cluster and the set of c representative points for the cluster, respectively. For a pair of points p, q, $dist(p, q)$ denotes the distance between the points. This distance could be any of the L_p metrics like L_1 ("manhattan") or L_2 ("euclidean") metrics. Alternatively, nonmetric *similarity functions* can also be used. The distance between two clusters u and v can then be defined as

$$dist(u, v) = \min_{p \in u.\text{rep}, q \in v.\text{rep}} dist(p, q)$$

For every cluster u, we keep track of the cluster closest to it in u.closest.

```
procedure merge(u, v)
begin
1.  w := u ∪ v
2.  w.mean := (|u| u.mean + |v| v.mean) / (|u| + |v|)
3.  tmpSet := ∅
4.  for i := 1 to c do {
5.     maxDist := 0
6.     foreach point p in cluster w do {
7.        if i = 1
8.           minDist := dist(p, w.mean)
9.        else
10.          minDist := min{dist(p, q) : q ∈ tmpSet}
11.       if (minDist ≥ maxDist){
12.          maxDist := minDist
13.          maxPoint := p
14.       }
15.    }
16.    tmpSet := tmpSet ∪ {maxPoint}
17. }
18. foreach point p in tmpSet do
19.    w.rep := w.rep ∪ {p + α*(w.mean-p) }
20. return w
end
```

Figure 6: Procedure for merging clusters

The algorithm makes extensive use of two data structures – a heap[CLR90] and a k-d tree[Sam90]. Furthermore, corresponding to every cluster, there exists a single entry in the heap – the entries for the various clusters u are arranged in the heap in the increasing order of the distances between u and u.closest. The second data structure is a k-d tree that stores the representative points for every cluster. The k-d tree is a data structure for efficiently storing and retrieving multi-dimensional point data. It is a binary search tree with the distinction that a different key value is tested at each level of the tree to determine the branch to traverse further. For example, for two dimensional data points, the first dimension of the point is tested at even levels (assuming the root node is level 0) while the second dimension is tested at odd levels. When a pair of clusters is merged, the k-d tree is used to compute the closest cluster for clusters that may previously have had one of the merged clusters as the closest cluster.

Clustering procedure: Initially, for each cluster u, the set of representative points u.rep contains only the point in the cluster. Thus, in Step 1, all input data points are inserted into the k-d tree. The procedure build_heap (in Step 2) treats each input point as a separate cluster, computes u.closest for each cluster u and then inserts each cluster into the heap (note that the clusters are arranged in the increasing order of distances between u and $u.closest$).

Once the heap Q and tree T are initialized, in each iteration of the while-loop, until only k clusters remain, the closest pair of clusters is merged. The cluster u at the top of the heap Q is the cluster for which u and $u.closest$ are the closest pair of clusters. Thus, for each step of the while-loop, extract_min (in Step 4) extracts the top element u in Q and also deletes u from Q. The merge procedure (see Figure 6) is then used to merge the closest pair of clusters u and v, and to compute new representative points for the new merged cluster w which are subsequently inserted into

T (in Step 8). The points in cluster w are simply the union of the points in the two clusters u and v that were merged. The merge procedure, in the for-loop (Steps 4-17), first iteratively selects c well-scattered points. In the first iteration, the point farthest from the mean is chosen as the first scattered point. In each subsequent iteration, a point from the cluster w is chosen that is farthest from the previously chosen scattered points. The points are then shrunk toward the mean by a fraction α in Step 19 of the merge procedure.

For the merged cluster w, since the set of representative points for it could have changed (a new set of representative points is computed for it), we need to compute its distance to every other cluster and set w.closest to the cluster closest to it (see Steps 11 and 12 of the cluster procedure). Similarly, for a different cluster x in Q, x.closest may change and x may need to be relocated in Q (depending on the distance between x and x.closest). A brute-force method for determining the closest cluster to x is to compute its distance with every other cluster (including w). However, this would require $O(n)$ steps for each cluster in Q, and could be computationally expensive and inefficient. Instead, we observe that the expensive computation of determining the closest cluster is not required for every cluster x. For the few cases that it is required, we use T to determine this efficiently in $O(\log n)$ steps per case. We can classify the clusters in Q into two groups. The first group of clusters are those who had either u or v as the closest cluster before u and v were merged. The remaining clusters in Q constitute the second group. For a cluster x in the first group, if the distance to w is smaller than its distance to the previously closest cluster (say u), then all we have to do is simply set w to be the closest cluster (see Step 17). The reason for this is that we know that the distance between x and every other cluster is greater than the distance between x and u. The problem arises when the distance between x and w is larger than the distance between x and u. In this case, any of the other clusters could become the new closest cluster to x. The procedure closest_cluster (in Step 15) uses the tree T to determine the closest cluster to cluster x. For every point p in x.rep, T is used to determine the nearest neighbor to p that is not in x.rep. From among the nearest neighbors, the one that is closest to one of x's representative points is determined and the cluster containing it is returned as the closest cluster to x. Since we are not interested in clusters whose distance from x is more than $dist(x, w)$, we pass this as a parameter to closest_cluster which uses it to make the search for nearest neighbors more efficient. Processing a cluster x in the second group is much simpler – x.closest already stores the closest cluster to x from among existing clusters (except w). Thus, if the distance between x and w is smaller than x's distance to it's previously closest cluster, x.closest, then w becomes the closest cluster to x (see Step 21); otherwise, nothing needs to be done. In case x.closest for a cluster x is updated, then since the distance between x and its closest cluster may have changed, x may need to be relocated in the heap Q (see Steps 18 and 22).

An improved merge procedure: In the merge procedure, the overhead of choosing representative points for the merged cluster can be reduced as follows. The merge procedure, in the outer for-loop (Step 4), chooses c scattered points from among all the points in the merged cluster w. Instead, suppose we selected the c scattered points for w from the $2c$ scattered points for the two clusters u and v being merged (the original scattered points for clusters u and v can be obtained by unshrinking their representative points by α). Then, since at most $2c$ points, instead of $O(n)$ points, need to be examined every time a scattered point is chosen, the complexity of the merge procedure reduces to $O(1)$. Furthermore, since the scattered points for w are chosen from the original scattered points for clusters u and v, they can be expected to be fairly well spread out.

3.3 Time and Space Complexity

The worst-case time complexity of our clustering algorithm can be shown to be $O(n^2 \log n)$. In [GRS97], we show that when the dimensionality of data points is small, the time complexity further reduces to $O(n^2)$. Since both the heap and the k-d tree require linear space, it follows that the space complexity of our algorithm is $O(n)$.

4 Enhancements for Large Data Sets

Most hierarchical clustering algorithms, including the one presented in the previous subsection, cannot be directly applied to large data sets due to their quadratic time complexity with respect to the input size. In this section, we present enhancements and optimizations that enable CURE to handle large data sets. We also address the issue of outliers and propose schemes to eliminate them.

4.1 Random Sampling

In order to handle large data sets, we need an effective mechanism for reducing the size of the input to CURE's clustering algorithm. One approach to achieving this is via *random sampling* – the key idea is to apply CURE's clustering algorithm to a random sample drawn from the data set rather than the entire data set. Typically, the random sample will fit in main-memory and will be much smaller than the original data set. Consequently, significant improvements in execution times for CURE can be realized. Also, random sampling can improve the quality of clustering since it has the desirable effect of filtering outliers.

Efficient algorithms for drawing a sample randomly from data in a file in one pass and using constant space are proposed in [Vit85]. As a result, we do not discuss sampling in any further detail, and assume that we employ one of the well-known algorithms for generating the random sample. Also, our experience has been that generally, the overhead of generating a random sample is very small compared to the time for performing clustering on the sample (the random sampling algorithm typically takes less than two seconds to sample a few thousand points from a file containing hundred thousand or more points).

Of course, one can argue that the reduction in input size due to sampling has an associated cost. Since we do not consider the entire data set, information about certain clusters may be missing in the input. As a result, our clustering algorithms may miss out certain clusters or incorrectly identify certain clusters. Even though random sampling does have this tradeoff between accuracy and efficiency, our experimental results indicate that for most of the data sets that we considered, with moderate sized random samples, we were able to obtain very good clusters. In addition, we can use chernoff bounds to analytically derive values for sample sizes for which the probability of missing clusters is low.

We are interested in answering the following question: what should the size s of the random sample be so that the probability of missing clusters is low ? One assumption that we will make is that the probability of missing a cluster u

is low if the sample contains at least $f|u|$ points from the sample, where $0 \le f \le 1$. This is a reasonable assumption to make since clusters will usually be densely packed and a subset of the points in the cluster is all that is required for clustering. Furthermore, the value of f depends on the cluster density as well as the intercluster separation – the more well-separated and the more dense clusters become, the smaller is the fraction of the points from each cluster that we need for clustering. *Chernoff bounds* [MR95] can be used to prove the following theorem.

Theorem 4.1: *For a cluster u, if the sample size s satisfies*

$$s \ge fN + \frac{N}{|u|}\log(\frac{1}{\delta}) + \frac{N}{|u|}\sqrt{(\log(\frac{1}{\delta}))^2 + 2f|u|\log(\frac{1}{\delta})} \quad (1)$$

then the probability that the sample contains fewer than $f|u|$ points belonging to cluster u is less than δ, $0 \le \delta \le 1$. ∎

Proof: See [GRS97]. ∎

Thus, based on the above equation, we conclude that for the sample to contain at least $f|u|$ points belonging to cluster u (with high probability), we need the sample to contain more than a fraction f of the total number of points – which seems intuitive. Also, suppose u_{min} is the smallest cluster that we are interested in, and s_{min} is the result of substituting $|u_{min}|$ for $|u|$ in the right hand side of Equation (1). It is easy to observe that Equation (1) holds for $s = s_{min}$ and all $|u| \ge |u_{min}|$. Thus, with a sample of size s_{min}, we can guarantee that with a high probability, $1-\delta$, the sample contains at least $f|u|$ points from an arbitrary cluster u. Also, assuming that there are k clusters, with a sample size of s_{min}, the probability of selecting fewer than $f|u|$ points from any one of the clusters u is bounded above by $k\delta$.

4.2 Partitioning for Speedup

As the separation between clusters decreases and as clusters become less densely packed, samples of larger sizes are required to distinguish them. However, as the input size n grows, the computation that needs to be performed by CURE's clustering algorithm could end up being fairly substantial due to the $O(n^2 \log n)$ time complexity. In this subsection, we propose a simple partitioning scheme for speeding up CURE when input sizes become large.

The basic idea is to partition the sample space into p partitions, each of size $\frac{n}{p}$. We then partially cluster each partition until the final number of clusters in each partition reduces to $\frac{n}{pq}$ for some constant $q > 1$. Alternatively, we could stop merging clusters in a partition if the distance between the closest pair of clusters to be merged next increases above a certain threshold. Once we have generated $\frac{n}{pq}$ clusters for each partition, we then run a second clustering pass on the $\frac{n}{q}$ partial clusters for all the partitions (that resulted from the first pass).

The idea of partially clustering each partition achieves a sort of *preclustering* – schemes for which were also proposed in [ZRL96]. The preclustering algorithm in [ZRL96] is incremental and scans the entire data set. Each successive data point becomes part of the closest existing cluster if it is within some threshold distance τ from it – else, it forms a new cluster. Thus, while [ZRL96] applies an incremental and approximate clustering algorithm to all the points, CURE uses it's hierarchical clustering algorithm only on the points in a partition. In some abstract sense, CURE's partitioning scheme behaves like a sieve working in a bottom-up fashion and filtering out individual points in favor of partial clusters that are then processed during the second pass.

The advantage of partitioning the input in the above mentioned fashion is that we can reduce execution times by a factor of approximately $\frac{q-1}{pq} + \frac{1}{q^2}$. The reason for this is that the complexity of clustering any one partition is $O(\frac{n^2}{p^2}(\frac{q-1}{q})\log\frac{n}{p})$ since the number of points per partition is $\frac{n}{p}$ and the number of merges that need to be performed for the number of clusters to reduce to $\frac{n}{pq}$ is $\frac{n}{p}(\frac{q-1}{q})$. Since there are p such partitions, the complexity of the first pass becomes $O(\frac{n^2}{p}(\frac{q-1}{q})\log\frac{n}{p})$. The time complexity of clustering the $\frac{n}{q}$ clusters in the second pass is $O(\frac{n^2}{q^2}\log\frac{n}{q})$. Thus, the complexity of CURE's partitioning algorithm is $O(\frac{n^2}{p}(\frac{q-1}{q})\log\frac{n}{p} + \frac{n^2}{q^2}\log\frac{n}{q})$, which corresponds to an improvement factor of approximately $\frac{q-1}{pq} + \frac{1}{q^2}$ over clustering without partitioning.

An important point to note is that, in the first pass, the closest points in each partition are merged only until the final number of clusters reduces to $\frac{n}{pq}$. By ensuring that $\frac{n}{pq}$ is sufficiently large compared to the number of desired clusters, k, we can ensure that even though each partition contains fewer points from each cluster, the closest points merged in each partition generally belong to the same cluster and do not span clusters. Thus, we can ensure that partitioning does not adversely impact clustering quality. Consequently, the best values for p and q are those that maximize the improvement factor $\frac{q-1}{pq} + \frac{1}{q^2}$ while ensuring that $\frac{n}{pq}$ is at least 2 or 3 times k.

The partitioning scheme can also be employed to ensure that the input set to the clustering algorithm is always in main-memory even though the random sample itself may not fit in memory. If the partition size is chosen to be smaller than the main-memory size, then the input points for clustering during the first pass are always main-memory resident. The problem is with the second pass since the size of the input is the size of the random sample itself. The reason for this is that for every cluster input to the second clustering pass, we store all the points in the cluster. Clearly, this is unnecessary, since our clustering algorithm only relies on the representative points for each cluster. Furthermore, the improved merge procedure in Section 3 only uses representative points of the previous clusters when computing the new representative points for the merged cluster. Thus, by storing only the representative points for each cluster input to the second pass, we can reduce the input size for the second clustering pass and ensure that it fits in main-memory.

4.3 Labeling Data on Disk

Since the input to CURE's clustering algorithm is a set of randomly sampled points from the original data set, the final k clusters involve only a subset of the entire set of points. In CURE, the algorithm for assigning the appropriate cluster labels to the remaining data points employs a fraction of *randomly* selected representative points for each of the final k clusters. Each data point is assigned to the cluster containing the representative point closest to it.

Note that approximating every cluster with multiple points instead a single centroid as is done in [ZRL96], enables CURE to, in the final phase, correctly distribute the data points when clusters are non-spherical or non-uniform. The final labeling phase of [ZRL96], since it employs only the centroids of the clusters for partitioning the remaining points, has a tendency to split clusters when they have non-spherical shapes or non-uniform sizes (since the space defined by a single centroid is a sphere).

4.4 Handling Outliers

Any data set almost always contains *outliers*. These do not belong to any of the clusters and are typically defined to be points of non agglomerative behavior. That is, the neighborhoods of outliers are generally sparse compared to points in clusters, and the distance of an outlier to the nearest cluster is comparatively higher than the distances among points in bonafide clusters themselves.

Every clustering method needs mechanisms to eliminate outliers. In CURE, outliers are dealt with at multiple steps. First, random sampling filters out a majority of the outliers. Furthermore, the few outliers that actually make it into the random sample are distributed all over the sample space. Thus, random sampling further isolates outliers.

In agglomerative hierarchical clustering, initially each point is a separate cluster. Clustering then proceeds by merging closest points first. What this suggests is that outliers, due to their larger distances from other points, tend to merge with other points less and typically grow at a much slower rate than actual clusters. Thus, the number of points in a collection of outliers is typically much less than the number in a cluster.

This leads us to a scheme of outlier elimination that proceeds in two phases. In the first phase, the clusters which are growing very slowly are identified and eliminated as outliers. This is achieved by proceeding with the clustering for some time until the number of clusters decreases below a certain fraction of the initial number of clusters. At this time, we classify clusters with very few points (e.g., 1 or 2) as outliers. The choice of a value for the fraction of initial clusters at which outlier elimination gets triggered is important. A very high value for the fraction could result in a number of cluster points being incorrectly eliminated – on the other hand, with an extremely low value, outliers may get merged into proper clusters before the elimination can kick in. An appropriate value for the fraction, obviously, is dependent on the data set. For most data sets we considered, a value of around $\frac{1}{3}$ performed well.

The first phase of outlier elimination is rendered ineffective if a number of outliers get sampled in close vicinity – this is possible in a randomized algorithm albeit with low probability. In this case, the outliers merge together preventing their elimination, and we require a second level of pruning. The second phase occurs toward the end. Usually, the last few steps of a clustering are the most important ones, because the granularity of the clusters is very high, and a single mistake could have grave consequences. Consequently, the second phase of outlier elimination is necessary for good clustering. From our earlier discussion, it is easy to observe that outliers form very small clusters. As a result, we can easily identify such small groups and eliminate them when there are very few clusters remaining, typically in the order of k, the actual number of clusters desired. We have found the above two-phase approach to outlier elimination to work very well in practice.

5 Experimental Results

In this section, we study the performance of CURE and demonstrate its effectiveness for clustering compared to BIRCH and MST[1] (*minimum spanning tree*). From our experimental results, we establish that

- BIRCH fails to identify clusters with non-spherical shapes (e.g., elongated) or wide variances in size.
- MST is better at clustering arbitrary shapes, but is very sensitive to outliers.
- CURE can discover clusters with interesting shapes and is less sensitive (than MST) to outliers.
- Sampling and partitioning, together, constitute an effective scheme for preclustering – they reduce the input size for large data sets without sacrificing the quality of clustering.
- The execution time of CURE is low in practice.
- Sampling and our outlier removal algorithm do a good job at filtering outliers from data sets.
- Our final labeling phase labels the data residing on disk correctly even when clusters are non-spherical.

In our experiments, in addition to CURE, we consider BIRCH and MST. We first show that, for data sets containing elongated or big and small clusters, both BIRCH and MST fail to detect the clusters while CURE discovers them with appropriate parameter settings. We then focus on analyzing the sensitivity of CURE to parameters like the shrink factor α and the random sample size s. Finally, we compare the execution times of BIRCH and CURE on a data set from [ZRL96], and present the results of our scale-up experiments with CURE. In all of our experiments, we use euclidean distance as the distance metric. We performed experiments using a Sun Ultra-2/200 machine with 512 MB of RAM and running Solaris 2.5.

Due to lack of space, we do not report all our experimental results – these can be found in [GRS97]. Also, the clusters in the figures in this section were generated using our labeling algorithm and visualizer. Our visualizer assigns a unique color to each cluster[2].

5.1 Algorithms

BIRCH: We used the implementation of BIRCH provided to us by the authors of [ZRL96]. The implementation performs preclustering and then uses a centroid-based hierarchical clustering algorithm with time and space complexity that is quadratic in the number of points after preclustering. We set parameter values to the default values suggested in [ZRL96]. For example, we set the page size to 1024 bytes and the input size to the hierarchical clustering algorithm after the preclustering phase to 1000. The memory used for preclustering was set to be about 5% of dataset size.

CURE: Our version of CURE is based on the clustering algorithm described in Section 3, that uses representative points with shrinking towards the mean. As described at the end of Section 3, when two clusters are merged in each step of the algorithm, representative points for the new merged cluster are selected from the ones for the two original clusters rather than all points in the merged cluster. This improvement speeds up execution times for CURE without adversely impacting the quality of clustering. In addition, we

[1] Clustering using MST is the same as the all-points approach described in Section 1, that is, hierarchical clustering using d_{min} as the distance measure.

[2] The figures in this paper were originally produced in color. These can be found in [GRS97].

Symbol	Meaning	Default Value	Range
s	Sample Size	2500	500 - 5000
c	Number of Representatives in Cluster	10	1 - 50
p	Number of Partitions	1	1 - 50
q	Reducing Factor for Each Partition	3	–
α	Shrink Factor	0.3	0 - 1.0

Table 1: Parameters

	Number of Points	Shape of Clusters	Number of Clusters
Data Set 1	100000	Big and Small Circles, Ellipsoids	5
Data Set 2	100000	Circles of Equal Sizes	100

Table 2: Data Sets

also employ random sampling, partitioning, outlier removal and labeling as described in Section 4. Our implementation makes use of the *k-d* tree and heap data structures. Thus, our implementation requires linear space and has a quadratic time complexity in the size of the input for lower dimensional data.

The partitioning constant q (introduced in Section 4.2) was set to 3. That is, we continue clustering in each partition until the number of clusters remaining is 1/3 of the number of points initially in the partition. We found that, for some of the data sets we considered, when q exceeds 3, points belonging to different clusters are merged, thus negatively impacting the clustering quality. Also we handle outliers as follows. First, at the end of partially clustering each partition, clusters containing only one point are considered outliers, and eliminated. Then, as the number of clusters approaches k, we again remove outliers – this time, the ones containing as many as 5 points. Finally, in almost all of our experiments, the random sample size was chosen to be about 2.5% of the initial data set size. Table 1 shows the parameters for our algorithm, along with their default values and the range of values for which we conducted experiments.

MST: When $\alpha = 0$ and the number of representative points c is a large number, CURE reduces to the MST method. Thus, instead of implementing the MST algorithm, we simply use CURE with the above parameter settings for α and c. This suffices for our purposes since we are primarily interested in performing qualitative measurements using MST.

5.2 Data sets

We experimented with four data sets containing points in two dimensions. Details of these experiments can be found in [GRS97]. Due to the lack of space, we report here the results with only two data sets whose geometric shape is as illustrated in Figure 7. The number of points in each data set is also described in Table 2. Data set 1 contains one big and two small circles that traditional partitional and hierarchical clustering algorithms, including BIRCH, fail to find. The data set also contains two ellipsoids which are connected by a chain of outliers. In addition, the data set has random outliers scattered in the entire space. The purpose of having outliers in the data set is to compare the sensitivity of CURE and MST to outliers. Data set 2 is identical to one of the data sets used for the experiments in [ZRL96]. It consists of 100 clusters with centers arranged in a grid pattern and data points in each cluster following a normal distribution with mean at the cluster center (a more detailed description of the data set can be found in [ZRL96]). We show that CURE not only correctly clusters this data set, but also that its execution times on the data set are much smaller than BIRCH.

5.3 Quality of Clustering

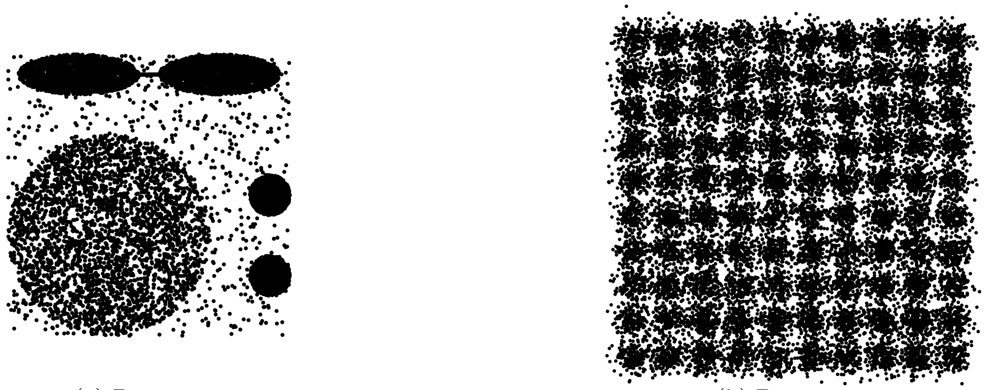
We run the three algorithms on Data set 1 to compare them with respect to the quality of clustering. Figure 8 shows the clusters found by the three algorithms for Data set 1. As expected, since BIRCH uses a centroid-based hierarchical clustering algorithm for clustering the preclustered points, it cannot distinguish between the big and small clusters. It splits the larger cluster while merging the two smaller clusters adjacent to it. In contrast, the MST algorithm merges the two ellipsoids because it cannot handle the chain of outliers connecting them. CURE successfully discovers the clusters in Data set 1 with the default parameter settings in Table 1, that is, $s = 2500$, $c = 10$, $\alpha = 0.3$ and $p = 1$. The moderate shrinking toward the mean by a factor of 0.3 enables CURE to be less sensitive to outliers without splitting the large and elongated clusters.

5.4 Sensitivity to Parameters

In this subsection, we perform a sensitivity analysis for CURE with respect to the parameters α, c, s and p. We use Data set 1 for our study. Furthermore, when a single parameter is varied, the default settings in Table 1 are used for the remaining parameters.

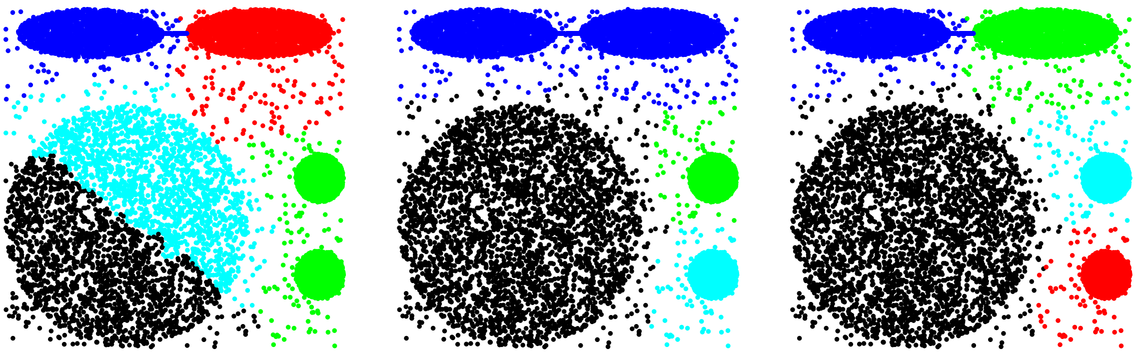
Shrink Factor α: Figure 9 shows the clusters found by CURE when α is varied from 0.1 to 0.9. The results, when $\alpha = 1$ and $\alpha = 0$, are similar to BIRCH and MST, respectively. These were presented in the previous subsection and thus, we do not present the results for these values of α. As the figures illustrate, when α is 0.1, the scattered points are shrunk very little and thus CURE degenerates to the MST algorithm which merges the two ellipsoids. CURE behaves similar to traditional centroid-based hierarchical algorithms for values of α between 0.8 and 1 since the representative points end up close to the center of the cluster. However, for the entire range of α values from 0.2 to 0.7, CURE always finds the right clusters. Thus, we can conclude that 0.2–0.7 is a good range of values for α to identify non-spherical clusters while dampening the effects of outliers.

Number of Representative Points c: We ran CURE while the number of representative points are varied from 1 to 100. For smaller values of c, we found that the quality of clustering suffered. For instance, when $c = 5$, the big cluster is split. This is because a small number of representative



(a) Data set 1 (b) Data set 2

Figure 7: Data sets



(a) BIRCH (b) MST METHOD (c) CURE

Figure 8: Data set 1

points do not adequately capture the geometry of clusters. However, for values of c greater than 10, CURE always found right clusters. In order to illustrate the effectiveness of the representative points in capturing the geometry of clusters, we plot, in Figure 10, the representatives for clusters at the end of clustering. The circular dot is the center of clusters and the representative points are connected by lines. As the figure illustrates, the representative points take shapes of clusters. Also, Figure 10(b) explains why the big cluster is split when we choose only 5 representative points per cluster. The reason is that the distance between the closest representative points belonging to the two subclusters of the large cluster becomes fairly large when c is 5.

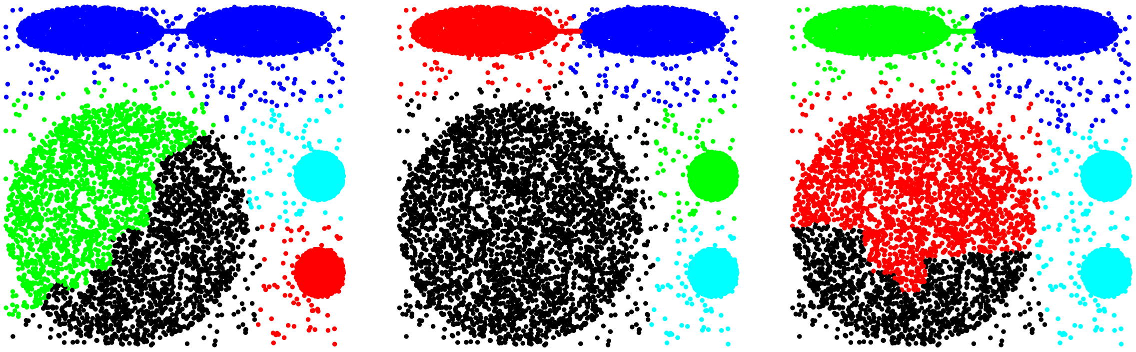
Number of Partitions p: We next varied the number of partitions from 1 to 100. With as many as 50 partitions, CURE always discovered the desired clusters. However, when we divide the sample into as many as 100 partitions, the quality of clustering suffers due to the fact that each partition does not contain enough points for each cluster. Thus, the relative distances between points in a cluster become large compared to the distances between points in different clusters – the result is that points belonging to different clusters are merged.

There is a correlation between the number of partitions p and the extent to which each partition is clustered as determined by q (a partition is clustered until the clusters remaining are $\frac{1}{q}$ of the original partition size). In order to preserve the integrity of clustering, as q increases, partition sizes must increase and thus, p must decrease. This is because with smaller partition sizes, clustering each partition to a larger degree could result in points being merged across clusters (since each partition contains fewer points from each cluster). In general, the number of partitions must be chosen such that $\frac{s}{pq}$ is fairly large compared to k (e.g., at least 2 or 3 times k).

Random Sample Size s: We ran CURE on Data set 1 with random sample sizes ranging from 500 upto 5000. For sample sizes up to 2000, the clusters found were of poor quality. However, from 2500 sample points and above (2.5% of the data set size), CURE always correctly identified the clusters.

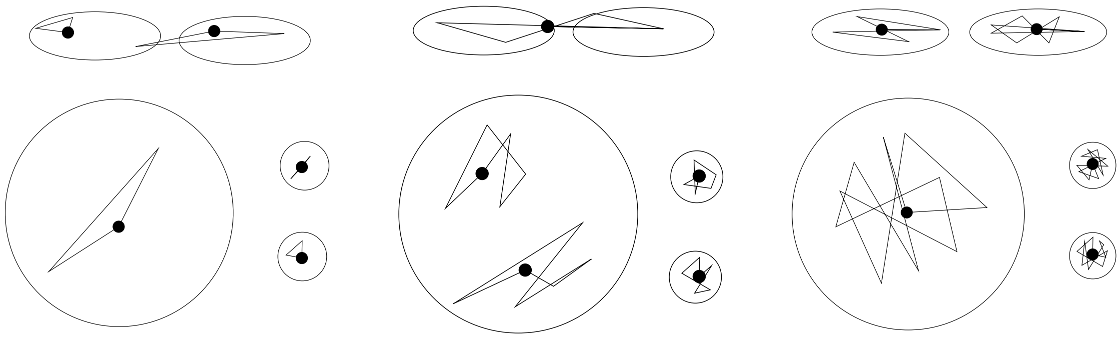
5.5 Comparison of Execution time to BIRCH

The goal of our experiment in this subsection is to demonstrate that using the combination of random sampling and partitioning to reduce the input size as is done by CURE can result in lower execution times than the preclustering scheme employed by BIRCH for the same purpose. We run both BIRCH and CURE on Data set 2 – this is the data set that centroid-based algorithms, in general, and BIRCH, in particular, can cluster correctly since it contains compact clusters with similar sizes. CURE, too, finds the right clusters with a random sample size of 2500, $\alpha = 1$ (which reduces CURE to a centroid-based algorithm), one representative for each cluster (that is, the centroid), and as many as 5 partitions.



(a) $\alpha = 0.1$ (b) $\alpha = 0.2 - 0.7$ (c) $\alpha = 0.8 - 0.9$

Figure 9: Shrink factor toward centroid



(a) $c = 2$ (b) $c = 5$ (c) $c = 10$

Figure 10: Representatives of clusters

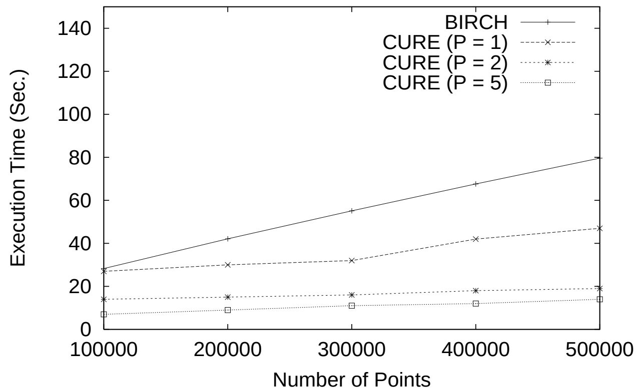


Figure 11: Comparison to BIRCH

Figure 11 illustrates the performance of BIRCH and CURE as the number of data points is increased from 100000 to 500000. The number of clusters or their geometry is not altered - thus, each cluster becomes more dense as the number of points increases. For CURE, we consider three values for the number of partitions: 1, 2 and 5, in order to show the effectiveness of partitioning. The execution times do not include the time for the final labeling phase since these are approximately the same for BIRCH and CURE, both of which utilize only the cluster centroid for the purpose of labeling.

As the graphs demonstrate, CURE's execution times are always lower than BIRCH's. In addition, partitioning further improves our running times by more than 50%. Finally, as the number of points is increased, execution times for CURE increase very little since the sample size stays at 2500, and the only additional cost incurred by CURE is that of sampling from a larger data set. In contrast, the executions times for BIRCH's preclustering algorithm increases much more rapidly with increasing data set size. This is because BIRCH scans the entire database and uses all the points in the data set for preclustering. Thus, the above results confirm that our proposed random sampling and partitioning algorithm are very efficient compared to the preclustering technique used in BIRCH.

5.6 Scale-up Experiments

The goal of the scale-up experiments is to determine the effects of the random sample size s, number of representatives c, number of partitions p and the number of dimensions on execution times. However, due to the lack of space, we only report our results when the random sample size is varied for Data set 1. The experimental results for varying number of representatives, partitions and dimensions can be found in [GRS97] (CURE took less than a minute to cluster points

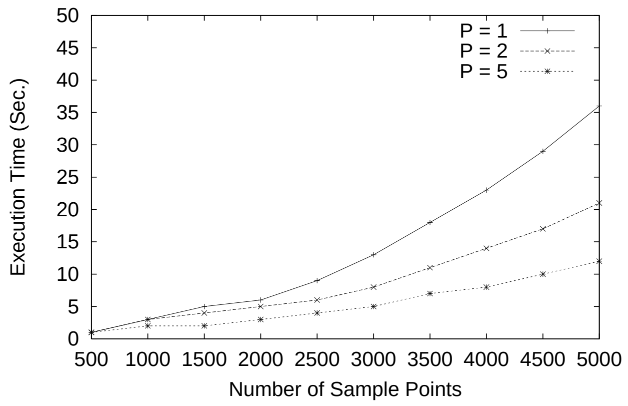


Figure 12: Scale-up experiments

with dimensionality as high as 40 – thus, CURE can comfortably handle high-dimensional data). In our running times, we do not include the time for the final labeling phase.

Random Sample Size: In Figure 12, we plot the execution time for CURE as the sample size is increased from 500 to 5000. The graphs confirm that the computational complexity of CURE is quadratic with respect to the sample size.

6 Concluding Remarks

In this paper, we addressed problems with traditional clustering algorithms which either favor clusters with spherical shapes and similar sizes, or are very fragile in the presence of outliers. We proposed a clustering method called CURE. CURE utilizes multiple representative points for each cluster that are generated by selecting well scattered points from the cluster and then shrinking them toward the center of the cluster by a specified fraction. This enables CURE to adjust well to the geometry of clusters having non-spherical shapes and wide variances in size. To handle large databases, CURE employs a combination of random sampling and partitioning that allows it to handle large data sets efficiently. Random sampling, coupled with outlier handling techniques, also makes it possible for CURE to filter outliers contained in the data set effectively. Furthermore, the labeling algorithm in CURE uses multiple random representative points for each cluster to assign data points on disk. This enables it to correctly label points even when the shapes of clusters are non-spherical and the sizes of clusters vary. For a random sample size of s, the time complexity of CURE is $O(s^2)$ for low-dimensional data and the space complexity is linear in s. To study the effectiveness of CURE for clustering large data sets, we conducted extensive experiments. Our results confirm that the quality of clusters produced by CURE is much better than those found by existing algorithms. Furthermore, they demonstrate that CURE not only outperforms existing algorithms but also scales well for large databases without sacrificing clustering quality.

Acknowledgments

We would like to thank Narain Gehani, Hank Korth, Rajeev Motwani and Avi Silberschatz for their encouragement and for their comments on earlier drafts of this paper. Without the support of Yesook Shim, it would have been impossible to complete this work.

References

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R^*-tree: an efficient and robust access method for points and rectangles. In *Proc. of ACM SIGMOD*, pages 322–331, Atlantic City, NJ, May 1990.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, Massachusetts, 1990.

[EKSX96] Martin Ester, Hans-Peter Kriegel, Jorg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial database with noise. In *Int'l Conference on Knowledge Discovery in Databases and Data Mining (KDD-96)*, Portland, Oregon, August 1996.

[EKX95] Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. A database interface for clustering in large spatial databases. In *Int'l Conference on Knowledge Discovery in Databases and Data Mining (KDD-95)*, Montreal, Canada, August 1995.

[GRS97] Sudipto Guha, R. Rastogi, and K. Shim. CURE: A clustering algorithm for large databases. Technical report, Bell Laboratories, Murray Hill, 1997.

[JD88] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data.* Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[NH94] Raymond T. Ng and Jiawei Han. Efficient and effective clustering methods for spatial data mining. In *Proc. of the VLDB Conference*, Santiago, Chile, September 1994.

[Ols93] Clark F. Olson. Parallel algorithms for hierarchical clustering. Technical report, University of California at Berkeley, December 1993.

[Sam89] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1989.

[Sam90] Hanan Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley Publishing Company, Inc., New York, 1990.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+ tree: a dynamic index for multi-dimensional objects. In *Proc. 13th Int'l Conference on VLDB*, pages 507–518, England, 1987.

[Vit85] Jeff Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.

[ZRL96] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: An efficient data clustering method for very large databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 103–114, Montreal, Canada, June 1996.