Finding Generalized Projected Clusters in High Dimensional Spaces

Charu C. Aggarwal, Philip S. Yu
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598
{ charu, psyu }@watson.ibm.com

Abstract

High dimensional data has always been a challenge for clustering algorithms because of the inherent sparsity of the points. Recent research results indicate that in high dimensional data, even the concept of proximity or clustering may not be meaningful. We discuss very general techniques for projected clustering which are able to construct clusters in arbitrarily aligned subspaces of lower dimensionality. The subspaces are specific to the clusters themselves. This definition is substantially more general and realistic than currently available techniques which limit the method to only projections from the original set of attributes. The generalized projected clustering technique may also be viewed as a way of trying to redefine clustering for high dimensional applications by searching for hidden subspaces with clusters which are created by inter-attribute correlations. We provide a new concept of using extended cluster feature vectors in order to make the algorithm scalable for very large databases. The running time and space requirements of the algorithm are adjustable, and are likely to tradeoff with better accuracy.

1 Introduction

The problem of clustering data points is defined as follows: Given a set of points in multidimensional space, find a partition of the points into *clusters* so that the points within each cluster are similar to one another. Various distance functions may be used in order to make a quantitative determination of similarity. In addition, an objective function may be defined with respect to this distance function in order to measure the overall quality of a partition. The method has been studied in considerable detail [5, 8, 15, 11, 17, 18] by both the statistics and database communities because of its applicability to many practical problems such as customer segmentation, pattern recognition, trend analysis and classification. An overview of clustering methods may be found in [11].

A common class of methods in clustering are *partitioning methods* in which a set of seeds (or representative objects) are used in order to partition the points implicitly [11]. Several variations of this technique exist such as the k-means and k-medoid algorithms [11]. In medoid-based techniques, the points from the database are used as seeds, as the algorithm tries to search for the optimal set of k seeds which results in the best clustering. An effective practical technique in this class called CLARANS [15] uses a restricted search space in order to improve efficiency.

Another well known class of techniques are *hierarchical clustering methods* in which the database is decomposed into several levels of partitioning which are represented by a dendogram [11]. Such methods are qualitatively effective, but practically infeasible for large databases since the performance is at least quadratic in the number of database points.

In density-based clustering methods [5], the neighborhood of a point is used in order to find dense regions in which clusters exist. Other related techniques for large databases include condensation and grid based methods in conjunction with spatial and hierarchical structures [8, 18]. The BIRCH method [18] uses a hierarchical data structure called the CF-Tree in order to incrementally build clusters. This is one of the most efficient approaches for low dimensional data, and it requires only one scan over the database. Another hierarchical method called CURE [8] was recently proposed, which tends to show excellent quality because it uses robust methods in order to measure distances between clusters. Therefore, it adjusts well to different shapes of clusters. An interesting grid-partitioning technique called *Optigrid* [9] has recently been proposed which is designed to perform well for high dimensional data.

In spite of these improved techniques, high dimensional data continues to pose a challenge to clustering algorithms at a very fundamental level. Most clustering algorithms do not work efficiently in higher dimensional

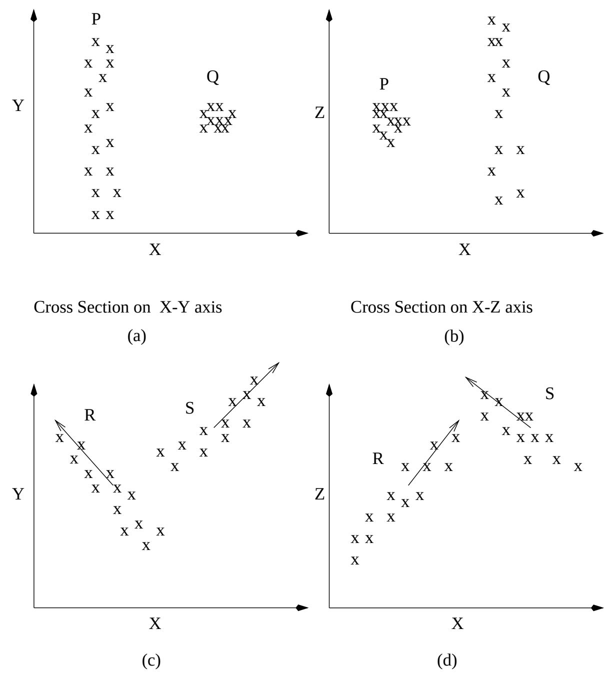

Figure 1: Illustrations of projected clustering

spaces because of the inherent sparsity of the data. This problem has been traditionally referred to as the *dimensionality curse* and is the motivation for a significant amount of work in high-dimensional clustering and similarity search [7, 9, 10, 13]. Recent theoretical results [3] have shown that in high dimensional space, the distance between every pair of points is almost the same for a wide variety of data distributions and distance functions. Under such circumstances, even the meaningfulness of proximity or clustering in high dimensional data may be called into question. One solution to this problem is to use feature selection in order to reduce the dimensionality of the space [14], but the correlations in the dimensions are often specific to data locality; in other words, some points are correlated with respect to a given set of features and others are correlated with respect to different features. Thus it may not always be feasible to prune off too many dimensions without at the same time incurring a substantial loss of information. We demonstrate this with the help of an example.

In Figure 1, we have illustrated four figures. The top two Figures 1(a) and 1(b) correspond to one set of points, while the bottom two Figures 1(c) and 1(d) correspond to another set. Figure 1(a) represents the X-Y cross-section of the first set of points, whereas Figure 1(b) represents the X-Z cross-section. There are two patterns of points labeled P and Q. The pattern P corresponds to a set of points which are close to one another in the X-Z plane. The second pattern Q corresponds to a set of points which are very close in the X-Y plane. Note that traditional feature selection does not work in this case, as each dimension is relevant to at least one of the clusters. At the same time, clustering in the full dimensional space will not discover the two patterns, since each of them is spread out along one of the dimensions. Figures 1(a) and (b) illustrate a simple case when the clusters are aligned along a particular axis system. Methods for finding such projected clusters have been discussed in [1]. Related methods for finding locally dense subspaces have also been discussed in [2, 4]. These are not really clustering methods, since they return dense rectangular projections with huge point overlaps. In reality, none of the above methods may be general enough in order to find the true clusters. The clusters could exist in arbitrarily oriented subspaces of lower dimensionality. The examples in Figures 1(c) and (d) illustrate these cases wherein the projected clusters could exist in arbitrarily oriented subspaces. The two such patterns are labeled by R and S. In this case, the planes on which the projection should take place are the normals to the arrows illustrated in the Figures 1(c) and 1(d). Often the examination of real data shows that points may tend to get aligned along arbitrarily skewed and elongated shapes in lower dimensional space because of correlations in the data. Clearly, the choice of axis parallel projections cannot find such clusters effectively.

In this context, we shall now define what we call a *generalized projected cluster*. A generalized projected cluster is a set $\mathcal{E}$ of vectors together with a set $\mathcal{C}$ of data points such that the points in $\mathcal{C}$ are closely clustered in the subspace defined by the vectors $\mathcal{E}$. The subspace defined by the vectors in $\mathcal{E}$ may have much lower dimensionality than the full dimensional space. The Figures 1(c) and (d) contain two clusters in arbitrarily oriented subspaces.

In this paper we focus on an algorithm to find clusters in lower dimensional projected subspaces for data of high dimensionality. We assume that the number k of clusters to be found is an input parameter. In addition to the number of clusters k the algorithm will take as input the dimensionality l of the subspace in which each cluster is reported. The output of the algorithm will be twofold:

- A $(k+1)$-way partition $\{\mathcal{C}_1, ..., \mathcal{C}_k, \mathcal{O}\}$ of the data, such that the points in each partition element except the last form a cluster. (The points in the last partition element are the *outliers*, which by definition do not cluster well.)
- A possibly different orthogonal set $\mathcal{E}_i$ of vectors for each cluster $\mathcal{C}_i$, $1 \leq i \leq k$, such that the points in $\mathcal{C}_i$ cluster well in the subspace defined by these vectors. (The vectors for the outlier set $\mathcal{O}$ can be assumed to be the empty set.) For each cluster $\mathcal{C}_i$, the cardinality of the corresponding set $\mathcal{E}_i$ is equal to the user defined parameter l.

In order to describe our algorithm we shall introduce a few notations and definitions. Let N denote the total number of data points. Assume that the dimensionality of the overall space is d. Let $\mathcal{C} = \{\overline{x_1}, \overline{x_2}, \ldots, \overline{x_t}\}$ be the set of points in a cluster. The *centroid* of a cluster is the algebraic average of all the points in the cluster. Thus, the centroid of the cluster $\mathcal{C}$ is given by $\overline{X}(\mathcal{C}) = \sum_{i=1}^{t} \overline{x_i}/t$. We will denote the distance between two points $\overline{x_1}$ and $\overline{x_2}$ by $dist(\overline{x_1}, \overline{x_2})$. In this paper, we will work with the euclidean distance metric.

Let $\overline{y} = (y_1, \ldots y_d)$ be a point in the d-dimensional space, and let $\mathcal{E} = \{\overline{e_1} \ldots \overline{e_l}\}$ be a set of $l \leq d$ orthonormal vectors in this d-dimensional space. These orthonormal vectors define a subspace. The projection $\mathcal{P}(\overline{y}, \mathcal{E})$ of point $\overline{y}$ in subspace $\mathcal{E}$ is an l-dimensional point given by $(\overline{y} \cdot \overline{e_1}, \ldots, \overline{y} \cdot \overline{e_l})$. Here $\overline{y} \cdot \overline{e_i}$ denotes the dot-product of $\overline{y}$ and $\overline{e_i}$.

Let $\overline{y_1}$ and $\overline{y_2}$ be two points in the original d-dimensional space. Then, the *projected distance* between the points $\overline{y_1}$ and $\overline{y_2}$ in subspace $\mathcal{E}$ is denoted by $Pdist(\overline{y_1}, \overline{y_2}, \mathcal{E})$ and is equal to the euclidean distance between the projections $\mathcal{P}(\overline{y_1}, \mathcal{E})$ and $\mathcal{P}(\overline{y_2}, \mathcal{E})$ in the l-dimensional space represented by $\mathcal{E}$. The *projected energy* of the cluster $\mathcal{C} = \{\overline{x_1}, \overline{x_2}, \ldots, \overline{x_t}\}$ in subspace $\mathcal{E}$ is denoted by $R(\mathcal{C}, \mathcal{E})$ and is given by the mean square distance of the points to the centroid of the cluster, when all points in $\mathcal{C}$ are projected to the subspace $\mathcal{E}$. Thus, we have:
$R(\mathcal{C}, \mathcal{E}) = \sum_{i=1}^{t} \{Pdist(\overline{x_i}, \overline{X}(\mathcal{C}), \mathcal{E})\}^2/t$

Note that the projected energy[1] of a cluster in a subspace is always less than that in the full dimensional space. For certain good choices of subspaces the projected energy is likely to be significantly smaller than that in full dimensional space. For example, for the case of Figures 1(c) and (d), the 2-dimensional subspaces in which the projected energy is likely to be small for clusters R and S are the planes normal to the arrows illustrated. It is the aim of the algorithm to discover clusters with small projected energy in subspaces of user-specified dimensionality l, where the subspace for each cluster could be different. Providing l as an input parameter gives the user considerable flexibility in discovering clusters in different dimensionalities. We will also discuss methods for providing the user guidance in finding a good value of l for which meaningful clusters may be found.

In this paper, we will examine this very general concept of using arbitrarily projected subspaces for finding clusters. Since dense full dimensional clusters cannot be found in very high dimensional data sets, this method searches for hidden subspaces in which points cluster well because of correlations among the dimensions. This technique can also be considered a meaningful re-definition of the clustering problem for very high dimensional data mining applications. We also propose an efficient algorithm for the projected clustering problem and make it scalable for vary large databases.

A very simplified version of this problem has been addressed in [1]. In this paper, a projected cluster is defined in terms of sets of points together with subsets of dimensions from the original data set. Such a framework may not necessarily be able to tackle the sparsity problem of high dimensional data because of the fact that real data often contains inter-attribute correlations. This naturally leads to projections which are not parallel to the original axis system. In fact, it is possible for data to continue to be very sparse in all possible projected subsets of attributes of the original data, yet have very dense projections in arbitrary directions. Our empirical results show that in such cases, axis-parallel projections are counter-productive since they lead to loss of information. Our paper provides a much more general framework and algorithms in which clusters can be represented in any arbitrarily projected space.

This paper is organized as follows. Section 2 describes our clustering algorithm in detail. The algorithm is known as ORCLUS (arbitrarily ORiented projected CLUSter generation). We discuss several improvements to the basic algorithm, and introduce the concept of *extended* cluster feature vectors (ECF) which are used to make the algorithm scale to very large databases. We also show how to use a progressive random sampling approach in order to improve the scalability with database size. In Section 3, we discuss the empirical results, whereas Section 4 discusses the conclusions and summary.

1.1 Projected Clusters and Skews

Most real data contains different kinds of skews in the data in which some subsets of dimensions are related to one another. Furthermore, these subsets of correlated dimensions may be different in different localities of the data. Correlated sets of dimensions lead to points getting aligned along arbitrary shapes in lower dimensional space. Such distributions of points which are far from the uniform distribution are referred to as skews. Projected clusters in lower dimensional subspaces are closely connected to the problem of finding skews in the data. Each orthogonal set of vectors which defines a subspace for a projected cluster provides a very good idea of the nature of skews and correlations in the data. For example, in the Figures 1(c) and (d), the arrows represent the directions of sparsity (greatest elongations because of correlated sets of dimensions). Coincidentally, the

[1] We have chosen the term energy, since this is a metric which is invariant on rotation of the axis system, and can be expressed as the sum of the energies of the individual axis directions.

direction of projection in which the points in the clusters are most similar are the normal planes to these arrows. In general, the subspace in which the maximum sparsity of point distribution occurs is the complementary subspace to the one in which the points are most similar to one another. For such distributions of points in very high dimensional space, the data is likely to continue to be very sparse in *full* dimensionality (thereby ruling out the use of full dimensional clustering algorithms such as those discussed in [17]), whereas the only realistic way of detecting regions of similarity would be to use lower dimensional projections for each cluster.

1.2 A Related Method

Singular Value Decomposition (SVD) [6, 16] is a well known technique in order to represent the data in a lower dimensional subspace by pruning away those dimensions which result in the least loss of information. Since the projected clustering method also projects the data into a lower dimensional subspace, it is interesting to examine the relationship between the two techniques. We will first give a brief overview of the methodology of singular value decomposition. The idea is to transform the data into a new coordinate system in which the (second order) correlations in the data are minimized. This transformation is done by using a two step process.

In the first step, the $d * d$ covariance matrix is constructed for the data set. Specifically, the entry (i, j) in the matrix is equal to the covariance between the dimensions i and j. The diagonal entries correspond to the variances of the individual dimension attributes. In the second step, the eigenvectors of this positive semi-definite matrix are found. These eigenvectors define an orthonormal system along which the second order correlations in the data are removed. The corresponding eigenvalues denote the spread (or variance) along each such newly defined dimension in this orthonormal system. Therefore, the eigenvectors for which the corresponding eigenvalues are the largest can be chosen as the subspace in which the data is represented. This results in only a small loss of information, since the data does not show much variance along the remaining dimensions.

The problem of dimensionality reduction is concerned with removing the dimensions in the *entire* data set so that the least amount of information is lost. On the other hand, the focus here is to find the best projection for each cluster in such a way that the greatest amount of *similarity* among the points in that cluster can be detected. Therefore, in the dimensionality reduction problem one chooses the eigenvectors with the *maximum* spread in order to retain the most information which distinguishes the points from one another. On the other hand, in the novel approach discussed in this paper, we pick the directions with the least spread for *each specific cluster*, so as to retain the information about the similarity of the points with one another in that cluster. Obviously, the directions with least spread are different for different clusters. The subspace which is complementary to the projection subspace for a given cluster is not useless information, since it may be used in order to retain the maximum information which distinguishes the points *within that cluster* from one another, if our technique is to be used for applications such as indexing.

The projected clustering problem is much more complex than the dimensionality reduction problem, because we are faced with the simultaneous problem of partitioning the data set as well as finding the directions with most similarity in each partition.

1.3 Covariance Matrix Diagonalization

In this section, we will discuss some properties of covariance matrix diagonalization which are useful for our clustering algorithm. We mention all the below properties as fact, which may be verified from [12]. Let C be the covariance matrix for the set of points $\mathcal{C}$. The matrix C is positive semidefinate and can be expressed in the format $C = P \Delta P^T$.

Here Δ is a diagonal matrix with non-negative entries. The columns in P are the (orthonormal) eigenvectors of C. The diagonal entries $\lambda_1 \ldots \lambda_d$ of Δ are the eigenvalues of C. The orthonormal eigenvectors define a new axis system and the matrix Δ is the covariance matrix of the original set of points when represented in this system. Since all non-diagonal entries are zero, it means that all second-order correlations have been removed. This also means that the eigenvalues correspond to the variances along each of the new set of axes. The trace $\sum_{i=1}^{d} \lambda_i$ is invariant under the axis-transformation defined by the eigensystem P and is equal to the trace of the original covariance matrix C. This is also equal to the energy of the cluster $\mathcal{C}$ in full dimensional space. It can also be shown that picking the $l \leq d$ smallest eigenvalues results in the l-dimensional subspace $\mathcal{E}$ of eigenvectors, in which sum of the variances along the l-directions is the least among all possible transformations. This is the same as the projected energy of the cluster $\mathcal{C}$ in subspace $\mathcal{E}$. *Thus, the diagonalization of the covariance matrix provides information about the projection subspace of a cluster which minimizes the corresponding energy.*

2 Generalized Projected Clustering

We decided on a variant of hierarchical merging methods for our algorithm. Unfortunately, the class of hierarchical methods is prohibitively expensive for very large databases, since the algorithms scale at least quadratically with the number of points. One solution is to run the method on only a random sample of points, but this can lead to a loss of accuracy. Consequently, we decided on the compromise solution of applying the technique to a small number k_0 of initial points, but using techniques from partitional clustering in order to always associate a current cluster with each of the points. The current clusters are then used in order to make more robust merging decisions. In effect, information from the entire database is used in the hierarchical merging process, even though the number of merges is greatly reduced. We will refer to the initial set of points as the *seeds*. At each stage of the algorithm, the following are associated with each seed s_i:

(1) **Current Cluster $\mathcal{C}_i$:** This is the set of points from the database which are closest to seed s_i in some subspace $\mathcal{E}_i$ associated with cluster $\mathcal{C}_i$. We assume that at each stage of the algorithm, the number of current clusters is denoted by k_c.

(2) **Current Subspace $\mathcal{E}_i$:** This is the subspace in which the points from $\mathcal{C}_i$ cluster well. The dimensionality l_c of $\mathcal{E}_i$ is at least equal to the user-specified dimensionality l. Initially, l_c is equal to the full dimensionality, and the value of l_c is reduced gradually to the user-specified dimensionality l. The idea behind this gradual reduction is that in the first few iterations the clusters may not necessarily correspond very well to the natural lower-dimensional subspace clusters in the data; and so a larger subspace is retained in order to avoid loss of information. (Only the most noisy subspaces are excluded.) In later iterations, the clusters are much more refined, and therefore subspaces of lower rank may be extracted.

The overall algorithm consists of a number of iterations, in each of which we apply a sequence of merging operations in order to reduce the number of current clusters by the factor $\alpha < 1$. We also reduce the dimensionality of current cluster $\mathcal{C}_i$ by $\beta < 1$ in a given iteration. Thus, the significance of dividing up the merging process over different iterations, is that each iteration corresponds to a certain dimensionality of the subspace in which the clusters are discovered. The first few iterations correspond to a higher dimensionality, and each successive iteration continues to peel off more and more noisy subspaces for the different clusters. The values of α and β need to be related in such a way that the reduction from k_0 to k clusters occurs in the same number of iterations as the reduction from $l_0 = |\mathcal{D}|$ to l dimensions. Therefore, the relationship $\log_{(1/\alpha)}(k_0/k) = \log_{(1/\beta)}(l_0/l)$ must be satisfied.

In the description of the algorithm ORCLUS, we have chosen $\alpha = 0.5$ and calculated the value of β according to the above relationship. The overall description of the algorithm is illustrated in Figure 2, and consists of a number of iterations in which each of the following three steps are applied:

```
Algorithm ORCLUS(Number of Clusters: k,
                 Number of Dimensions: l)
{ $\mathcal{C}_i$ is the current cluster $i$ }
{ $\mathcal{E}_i$ is the set of vectors defining subspace for cluster $\mathcal{C}_i$ }
{ $k_c \Rightarrow$ current number of seeds; $l_c \Rightarrow$ current
             dimensionality associated with each seed }
{ $S = \{s_1, s_2 \ldots s_{k_c}\}$ is the current set of seeds }
{ $k_0$ is the number of seeds that we begin with }
begin
  Pick $k_0 > k$ points from the database and denote by
     $S$; $\{S = (s_1, \ldots s_{k_c})$ }
  $k_c = k_0$; $l_c = d$;
  for each $i$ set $\mathcal{E}_i = \mathcal{D}$;
  { Initially, $\mathcal{E}_i$ is the original axis-system }
  $\alpha = 0.5$; $\beta = e^{-\log(d/l)\cdot\log(1/\alpha)/\log(k_0/k)}$;
  while $(k_c > k)$ do
  begin
    { Find partitioning induced by the seeds }
    $(s_1, \ldots s_{k_c}, \mathcal{C}_1, \ldots \mathcal{C}_{k_c}) = Assign(s_1, \ldots s_{k_c}, \mathcal{E}_1, \ldots \mathcal{E}_{k_c})$;
    { Determine current subspace associated with
             each cluster $\mathcal{C}_i$ }
    for $i = 1$ to $k_{new}$ do $\mathcal{E}_i = FindVectors(\mathcal{C}_i, l_{new})$;
    { Reduce number of seeds and dimensionality
                associated with each seed }
    $k_{new} = \max\{k, k_c \cdot \alpha\}$; $l_{new} = \max\{l, l_c \cdot \beta\}$;
    $(s_1 \ldots s_{k_{new}}, \mathcal{C}_1, \ldots \mathcal{C}_{k_{new}}, \mathcal{E}_1 \ldots \mathcal{E}_{k_{new}}) =$
             $Merge(\mathcal{C}_1, \ldots \mathcal{C}_{k_c}, k_{new}, l_{new})$;
    $k_c = k_{new}$; $l_c = l_{new}$;
  end;
  $(s_1, \ldots s_k, \mathcal{C}_1, \ldots \mathcal{C}_k) = Assign(s_1, \ldots s_k, \mathcal{E}_1, \ldots \mathcal{E}_k)$;
  return$(\mathcal{C}_1 \ldots \mathcal{C}_{k_c})$;
end;
```

Figure 2: The Clustering Algorithm

```
Algorithm Assign($s_1, \ldots s_{k_c}$, $\mathcal{E}_1 \ldots \mathcal{E}_{k_c}$)
begin
  for each $i \in \{1, \ldots, k_c\}$ do $\mathcal{C}_i = \phi$;
  for each data point $p$ do begin
    Determine $Pdist(p, s_i, \mathcal{E}_i)$ for each $i \in \{1, \ldots, k_c\}$
    { Distance of point $p$ to $s_i$ in subspace $\mathcal{E}_i$};
    Determine the seed $s_i$ with the least value of
    $Pdist(p, s_i, \mathcal{E}_i)$ and add $p$ to $\mathcal{C}_i$;
    end
  for each $i \in \{1, \ldots, k_c\}$ do $s_i = \overline{X}(\mathcal{C}_i)$;
  return$(s_1, \ldots s_{k_c}, \mathcal{C}_1 \ldots \mathcal{C}_{k_c})$;
end
```

Figure 3: Creating the Cluster Partitions

```
Algorithm FindVectors(Cluster of points:$\mathcal{C}$,
            Dimensionality of projection:$q$)
begin
  Determine the $d * d$ covariance matrix $\mathcal{M}$ for $\mathcal{C}$;
  Determine the eigenvectors of matrix $\mathcal{M}$;
  $\mathcal{E}$ = Set of eigenvectors corresponding to
           smallest $q$ eigenvalues;
  return$(\mathcal{E})$;
end
```

Figure 4: Finding the Best Subspace for a Cluster

```
Algorithm $Merge(s_1 \ldots s_{k_c}, k_{new}, l_{new})$
begin
    for each pair $i, j \in \{1, \ldots k_c\}$ satisfying $i < j$ do begin
        $\mathcal{E}'_{ij} = FindVectors(\mathcal{C}_i \cup \mathcal{C}_j, l_{new})$;
        { Defined by eigenvectors for $l_{new}$ smallest eigenvalues }
        $s'_{ij} = \overline{X}(\mathcal{C}_i \cup \mathcal{C}_j)$ { Centroid of $\mathcal{C}_i \cup \mathcal{C}_j$ };
        $r_{ij} = R(\mathcal{C}_i \cup \mathcal{C}_j, \mathcal{E}'_{ij})$
            { Projected energy of $\mathcal{C}_i \cup \mathcal{C}_j$ in subspace $\mathcal{E}'_{ij}$ }
    end
    while $(k_c > k_{new})$ do begin
        Find the smallest value of $r_{i'j'}$ among
            all pairs $i, j \in \{1, \ldots k_c\}$ satisfying $i < j$;
        { Merge the corresponding clusters $\mathcal{C}_{i'}$ and $\mathcal{C}_{j'}$; }
        $s_{i'} = s'_{i'j'}$; $\mathcal{C}_{i'} = \mathcal{C}_{i'} \cup \mathcal{C}_{j'}$; $\mathcal{E}_{i'} = \mathcal{E}'_{i'j'}$;
        Discard seed $s_{j'}$ and $\mathcal{C}_{j'}$ and renumber the
           seeds/clusters indexed larger than $j'$ by subtracting 1;
        Renumber the values $s'_{ij}$, $\mathcal{E}_{ij}$, $r_{ij}$ correspondingly
                for any $i, j \geq j'$;
        { Since cluster $i'$ is new, the pairwise recomputation
          for $r_{i'j}$ for different $j$ needs to be done }
        for each $j \neq i' \in \{1, \ldots k_c - 1\}$ do begin
            Recompute $\mathcal{E}'_{i'j} = FindVectors(\mathcal{C}_{i'} \cup \mathcal{C}_j, l_{new})$;
            $s'_{i'j} = \overline{X}(\mathcal{C}_{i'} \cup \mathcal{C}_j)$; { Centroid of $\mathcal{C}_{i'} \cup \mathcal{C}_j$ }
            $r_{i'j} = R(\mathcal{C}_{i'} \cup \mathcal{C}_j, \mathcal{E}'_{i'j})$ { Proj. En. of $\mathcal{C}_{i'} \cup \mathcal{C}_j$ in $\mathcal{E}'_{i'j}$; }
        end
    $k_c = k_c - 1$;
    end
    return$(s_1, \ldots s_{k_{new}}, \mathcal{C}_1, \ldots \mathcal{C}_{k_{new}}, \mathcal{E}_1, \ldots \mathcal{E}_{k_{new}})$
end
```

Figure 5: The Merging Algorithm

(1) **Assign:** The database is partitioned into k_c current clusters by assigning each point to its closest seed. In the process of partitioning, the distance of a database point to seed s_i is measured in the subspace $\mathcal{E}_i$. In other words, for each database point p, the value of $Pdist(p, s_i, \mathcal{E}_i)$ is computed, and the point p is assigned to the current cluster $\mathcal{C}_i$ for which this value is the least. At the end of this procedure each seed is replaced by the centroid of the cluster which was just created. The procedure serves to refine both the set of clusters in a given iteration and the set of seeds associated with these clusters. This method is illustrated in Figure 3.
(2) **FindVectors:** In this procedure we find the subspace $\mathcal{E}_i$ of dimensionality l_c for each current cluster $\mathcal{C}_i$. This is done by computing the covariance matrix for the cluster $\mathcal{C}_i$ and picking the l_c orthonormal eigenvectors with the least spread (eigenvalues). This finds the least energy subspace of rank l_c for cluster $\mathcal{C}_i$. The value of l_c reduces from iteration to iteration. The overall process is illustrated in Figure 4.
(3) **Merge:** During a given iteration, the *Merge* phase reduces the number of clusters from k_c to $k_{new} = (1-\alpha) \cdot k_c$. In order to do so, closest pairs of current clusters need to be merged successively. This is easy to do in full dimensional algorithms, since the goodness of the merge can be easily quantified in full dimensionality. This case is somewhat more complex. Since each current cluster $\mathcal{C}_i$ exists in its own (possibly different) subspace $\mathcal{E}_i$, how do we decide the suitability of merging "closest" pairs? Since, the aim of the algorithm is to discover clusters with small projected energy, we design a measure for testing the suitability of merging two clusters by examining the projected energy of the union of the two clusters in the corresponding least spread subspace. The quantitative measure for the suitability of merging each pair of seeds $[i, j]$ is calculated using a two step process.

In the first step, use singular value decomposition on the points in $\mathcal{C}_i \cup \mathcal{C}_j$ and find the eigenvectors corresponding to the smallest l_{new} eigenvalues. (l_{new} is the projected dimensionality in that iteration.) These eigenvectors define the least spread subspace for the points in the union of the two segmentations. Let us denote this subspace by $\mathcal{E}'_{ij}$. In the second step, we find the centroid s'_{ij} of $\mathcal{C}_i \cup \mathcal{C}_j$ and use the energy $r_{ij} = R(\mathcal{C}, \mathcal{E}'_{ij})$ of this cluster in the subspace $\mathcal{E}'_{ij}$ as the indicator of how well the points for the two clusters combine into a single cluster. Note that the points for the two clusters are likely to combine well using this method, if the least spread directions for the individual clusters were similar to begin with. In this case, $\mathcal{E}'_{ij}$, $\mathcal{E}_i$, and $\mathcal{E}_j$ are all likely to be similarly oriented subspaces with small (projected) energies. The overall idea here is to measure how well the two current clusters can be fit into a single pattern of behavior. For example, in the Figure 1(c), two current clusters $\mathcal{C}_i$ and $\mathcal{C}_j$ which consist of points from the same data pattern (say R) are likely to result in similar planes of 2-dimensional projection (normal plane to the direction of the arrows for R). Consequently, the points in $\mathcal{C}_i \cup \mathcal{C}_j$ are likely to be projected to a similar plane. The value of the energy r_{ij} when measured in this plane is likely to be small.

The above two-step process is repeated for each pair of seeds, and the pair of seeds $[i', j']$ is found for which $r_{i'j'}$ is the least. If the seeds are merged, then the centroid $s'_{i'j'}$ of the combined cluster is added to the set of seeds, whereas seeds $s_{i'}$ and $s_{j'}$ are removed. The current cluster associated with this new seed $s_{i'j'}$ is $\mathcal{C}_{i'} \cup \mathcal{C}_{j'}$, and the current subspace is $\mathcal{E}'_{ij}$. This agglomeration procedure is repeated multiple times, so that the number of clusters is reduced by a factor of α. The overall merging procedure is described in Figure 5.

The algorithm terminates when the merging process over all the iterations has reduced the number of clusters to k. At this point, the dimensionality l_c of the subspace $\mathcal{E}_i$ associated with each cluster $\mathcal{C}_i$ is also equal to l. The algorithm performs one final pass over the database in which it uses the *Assign* procedure in order to partition the database. If desired, the *FindVectors()* procedure may be used in order to determine the optimum subspace associated with each cluster at termination.

One of the aspects of this merging technique is that it needs to work explicitly with the set of current clusters $\mathcal{C}_1 \ldots \mathcal{C}_{k_c}$. Since the database size may be very large, it would be extremely cumbersome to maintain the sets $\mathcal{C}_1 \ldots \mathcal{C}_{k_c}$ explicitly. Furthermore, the covariance matrix calculation is also likely to be very I/O intensive, since each set of covariance matrix calculations is likely to require a pass over the database. Since the covariance matrix may be calculated $O(k_0^2)$ times by the *Merge* operation (see analysis later), this translates to $O(k_0^2)$ passes over the database. The value of k_0 is likely to be several times the number of clusters k to be determined. This level of I/O is unacceptable. We introduce the concept of *extended cluster feature vectors*, so that all of the above operations can be performed by always maintaining certain summary information about the clusters. This provides considerable ease in calculation of the covariance matrices. Details of this technique will be provided in a later section.

2.1 Picking the Projected Dimensionality

An important input parameter to the algorithm is the projected dimensionality l. To give some guidance to the user in picking this parameter, we design a measure which is motivated by the reason we have defined the concept of generalized projected clustering in the first place. It has been proved in [3], that with increasing dimensionality the distance between every pair of points is almost the same under certain conditions on the data distributions. This means that if a cluster of points $\mathcal{C}$ is compared to the universal set of points $\mathcal{U}$ in very high dimensional space $\mathcal{D}$, then we have $\frac{R(\mathcal{C},\mathcal{D})}{R(\mathcal{U},\mathcal{D})} \approx 1$.

This is a very undesirable situation because it indicates that the average spread (energy) for a cluster is almost the same as the average spread for the points in the entire database in full dimensional space. This is also the reason for the instability of randomized clustering algorithms in high dimensional space: different runs lead to different clusters, all of which are almost equally good according to this or other measures.

However, in subspaces of $\mathcal{D}$, this ratio may be much smaller than 1. In general a ratio which is significantly smaller than 1 is desirable, because it indicates a tightly knit cluster in the corresponding projected subspace. We define the following quality measure called *cluster sparsity coefficient*: $S(\mathcal{C}_1 \ldots \mathcal{C}_k, \mathcal{E}_1 \ldots \mathcal{E}_k) = (1/k) \cdot \sum_{i=1}^{k} \frac{R(\mathcal{C}_i,\mathcal{E}_i)}{R(\mathcal{U},\mathcal{E}_i)}$. The lower the value of l, the smaller the fraction is likely to be, because $\mathcal{E}_i$ may be picked in a more optimum way in lower dimensionality so as to reduce the energy $R(\mathcal{C}_i, \mathcal{E}_i)$ for the particular distribution of points in $\mathcal{C}_i$, whereas the aggregate set of points in $\mathcal{U}$ may continue to have high energy $R(\mathcal{U}, \mathcal{E}_i)$. At termination, the algorithm may return the cluster sparsity coefficient. If this value is almost 1, then it is clear that a smaller value of the projected dimensionality needs to be picked. In fact the user may define a minimum threshold for this (intuitively interpretable) quality measure and pick the largest value of l at which the cluster sparsity coefficient returned at termination is less than the threshold.

2.2 Outlier Handling

In order to take into account the fact that some of the points are outliers, we may need to make some modifications to the algorithm. Let δ_i be the projected distance of the nearest other seed to the seed s_i in subspace $\mathcal{E}_i$ for each $i \in \{1, \ldots k_c\}$. Consider an arbitrary point P in the database during the assignment phase. Let s_r be the seed to which the database point P is assigned during the assignment phase. The point P is an outlier, if its projected distance to seed s_r in subspace E_r is larger than δ_r.

In addition, some of the seeds which were initially chosen may also be outliers. These need to be removed during the execution of the algorithm. A simple modification which turns out to be quite effective is the discarding of a certain percentage of the seeds in each iteration, for which the corresponding clusters contain very few points. When the outlier handling option is implemented, the value of the seed reduction factor α is defined by the percentage reduction in each iteration due to either merges or discards.

2.3 Scalability for Very Large Databases

As discussed above, the times for calculating covariance matrices can potentially be disastrously large (especially in terms of I/O times), if the covariance matrix is calculated from scratch for each $FindVectors(\cdot)$ operation. Therefore, we use a concept similar to the use of the Cluster Feature vector (CF-vector) [18] in order to find the covariance matrix efficiently. We shall refer to this as the extended CF-vector (or ECF-vector).

The ECF-vector is specific to a given cluster $\mathcal{C}$, and contains $d^2 + d + 1$ entries. These entries are of three kinds:

(1) There are d^2 entries corresponding to each pair of dimensions (i, j). For each pair of dimensions (i, j), we sum the products of the ith and jth components for each point in the cluster. In other words, of x_i^k denotes the ith component of the kth point in the cluster, then for a cluster $\mathcal{C}$ and pair of dimensions (i, j), we maintain the entry $\sum_{k \in \mathcal{C}} x_i^k \cdot x_j^k$. Thus there are d^2 entries of this type. For a cluster $\mathcal{C}$, we will refer to this entry as $ECF1_{ij}^{\mathcal{C}}$ for each pair of dimensions i and j. We will refer to the entire set of such entries as $\overline{ECF1^{\mathcal{C}}}$.

(2) There are d entries corresponding to each dimension i. We sum the ith components for each point in the cluster. Thus, we maintain the entry $\sum_{k \in \mathcal{C}} x_i^k$. For a cluster $\mathcal{C}$, we shall refer to this entry as $ECF2_i^{\mathcal{C}}$. The

corresponding set of entries is referred to as $\overline{ECF2^{\mathcal{C}}}$.
(3) The number of points in the cluster $\mathcal{C}$ is denoted by $ECF3^{\mathcal{C}}$.

Note that the first set of d^2 entries are not included in the standard definition of CF-vector as introduced in [18]. The entire cluster feature vector is denoted by $\overline{ECF^{\mathcal{C}}} = (\overline{ECF1^{\mathcal{C}}}, \overline{ECF2^{\mathcal{C}}}, ECF3^{\mathcal{C}})$. Two important features of the ECF-vector are as follows:

Observation 2.1 *The covariance matrix can be derived directly from the ECF-vector. Specifically, the covariance between the dimensions i and j for a set of points $\mathcal{C}$ is equal to $ECF1^{\mathcal{C}}_{ij}/ECF3^{\mathcal{C}} - ECF2^{\mathcal{C}}_i \cdot ECF2^{\mathcal{C}}_j/(ECF3^{\mathcal{C}})^2$.*

Proof: The average of the product of the ith and jth attribute is given by $ECF1^{\mathcal{C}}_{ij}/ECF3^{\mathcal{C}}$, whereas the average of the ith attribute is given by $ECF2^{\mathcal{C}}_i/ECF3^{\mathcal{C}}$. Since the covariance between the ith and jth attributes is given by the subtraction of the product of averages from the average product, the result follows. ■
It has been established by Zhang et. al. [18] that the CF-vector may be used in order to calculate the centroid, and radius of a cluster. Since the ECF-vector is a superset of the CF-vector, these measures may also be calculated from the ECF-vector. Our use of the extended CF-vector provides the ability to calculate the covariance matrix as well. The usefulness of the above result is that the covariance matrix can be calculated very efficiently from the ECF-vector. The use of the summary characteristics of a cluster is so useful because of the fact that it satisfies the additive property.

Observation 2.2 *The ECF-vector satisfies the additive property. The ECF-vector for the union of two sets of points is equal to the sum of the corresponding ECF-vectors.*

The proof of the above trivially follows from the fact that the ECF-vector for a set of points can be expressed as the sum of the ECF-vectors of the individual points. The additive property ensures that while constructing a ECF-vector for the union of two clusters, it is not necessary to recalculate everything from scratch, but that it is sufficient to add the ECF-vectors of the two clusters.

The ECF-vectors are used in order to modify ORCLUS in the following way. *During the entire operation of the algorithm, the current clusters $\mathcal{C}_1 \ldots \mathcal{C}_{k_c}$ associated with each seed are not maintained explicitly.* Instead, the extended CF-vectors for the seeds are maintained. The ECF-vector for a cluster is sufficient to calculate the radius, centroid [18], and covariance matrix for each cluster. In each iteration, the seeds are defined by the centroids of the current clusters for the last iteration. In addition, the additive property of the ECF-vectors ensures that during a *Merge* operation, the ECF-vector for the merged cluster may be calculated easily. The ECF-vectors need to be recalculated in each iteration only during the *Assign(·)* phase; a simple additive process which does not affect the overall time-complexity of the algorithm.

2.4 Running Time Requirements

The running time of the algorithm depends upon the initial number of seeds k_0 chosen by the algorithm. The skeletal structure for the algorithm contains the two basic process of merging and assignment of points to seeds. (The addition of the outlier handling option only reduces the running time of the algorithm since merges are replaced by simple discards. Therefore the analysis presented below is an overestimate on the running time of the algorithm.) The running time for the various procedures of the algorithm are as below:

(1) Merge: The time for merging is asymptotically controlled by the time required in the first iteration of reducing the value of the number of current clusters from k_0 to $\alpha \cdot k_0$. To start off, the eigenvectors for each of the k_0^2 pairs of current clusters are calculated. This requires a running time of $O(d^3)$ for each pair. Furthermore, for each of the subsequent (at most) $(1-\alpha)\cdot k_0 - k$ merges, the eigenvectors for $O(k_0)$ pairs of clusters need to be re-calculated. Since, each eigenvector calculation is $O(d^3)$, it follows that the total time for eigenvector calculations during the merge operation of the algorithm is given by $O(k_0^2 \cdot d^3)$. Furthermore, for each merge operation, $O(k_0^2)$ time is required in order to pick the cluster with the least energy. Therefore, the total running time for all merges is given by $O(k_0^2(k_0 + d^3))$.
(2) Assign: The running time for assignment of the N points in d-dimensional space to the k_0 clusters in the first iteration is given by $O(k_0 \cdot N \cdot d)$. In the second iteration, the time is given by $O((k_0 \cdot \alpha) \cdot N \cdot d)$, and so on. Therefore the overall running time for the assignment process is given by $O(k_0 \cdot N \cdot d/(1-\alpha))$.
(3) Subspace Determination: The time for eigenvector calculations (or subspace determinations) during the *Merge* phase has already been included in the analysis above. It now only remains to calculate the time for subspace determinations during each iterative phase (the *FindVectors* procedure after the *Assign*) of the algorithm. During the first iteration of the algorithm, there are k_0 subspace determinations, during the second iteration, there are $k_0 \cdot \alpha$ subspace determinations and so on. Therefore, during the entire algorithm, there are at most $k_0/(1-\alpha)$ subspace determinations. This running time is strictly dominated by the subspace determinations during the *Merge* phase, and can hence be ignored asymptotically.

Upon summing up the various components of the running time, the total time required is given by $O(k_0^3 + k_0 \cdot N \cdot d + k_0^2 \cdot d^3)$. Note that this running time is dependent upon the choice of the initial parameter k_0. A choice of a larger value of k_0 is likely to increase the running time, but also improve the quality of clustering.

2.5 Space Requirements

The use of the ECF vector cuts down the space requirements of the algorithm considerably, since the current clusters associated with each seed need not be maintained. During each iteration of the algorithm, at most k_c extended CF-vectors need to be maintained. The space requirement for this is given by $k_c \cdot (d^2+d+1)$. Since $k_c \leq k_0$, the overall space requirement of the algorithm is $O(k_0 \cdot d^2)$. This is independent of database size, and may very easily fit in main memory for reasonable values of k_0 and d.

2.6 Progressive Sampling Techniques

It is possible to speed up the assignment phase of the algorithm substantially by using a progressive type of random sampling. Note that the component in the running time which is dependent on N is caused by the *Assign(·)* procedure of the algorithm. This component of the running time can be the bottleneck when the database is very large. Another observation is that the CPU time for performing the *Assign(·)* procedure reduces by the factor α in each iteration, as the number of seeds reduces by the same factor. The purpose of the *Assign(·)* procedure is to keep correcting the seeds in each iteration so that each seed is a more central point of the current cluster that is associated with it. It is possible to achieve the same results by using a progressive kind of random sampling. Thus, only a randomly sampled subset of the points are assigned to the seeds in each iteration. This random sample can be different over multiple iterations and can change in size. (If the size of the random sample increases by a factor of α in each iteration, then the time for the assignment phase is the same in each iteration.) Thus, one possibility is to start off with a random sample of size $N \cdot k/k_0$ and increase by a factor of α in each iteration, so that in the final iteration all the database points are used. Thus, the time for the assignment phase in each iteration would be $N \cdot k \cdot d$, and over all iterations would be $N \cdot k \cdot d \cdot \log_\alpha(k_0/k)$. This kind of progressive sampling can provide considerable savings in CPU time in the first few iterations, if k_0 is substantially larger than k. Furthermore, it is unlikely to lose much in terms of accuracy because when the seeds become more refined in the last few iterations, then larger sample sizes are used.

3 Empirical Results

The simulations were performed on a 233-MHz IBM RS/6000 computer with 128 M of memory, 2GB SCSI drive, and running AIX 4.1.4. We tested the following accuracy and performance measures:

1. Accuracy of the clustering with respect to matching of points in Input and Output Clusters.
2. Scaling of running times with database size.
3. Scaling of the I/O performance with initial number of seeds.

In order to test the accuracy results, we determined the *Confusion Matrix* which indicated how well the output clusters matched with the input points. The entry (i, j) of the confusion matrix indicates the number of points belonging to the output cluster i, which were generated as a part of the input cluster j. If the clustering algorithm performs well, then each row and column is likely to have one entry which is significantly larger than the others. On the other hand, in the case when the clustering technique is so bad as to be completely random, then the points are likely to be evenly distributed among the different clusters.

3.1 Synthetic Data Generation

The first stage of synthetic data generation was to find the arbitrarily oriented subspaces associated with each cluster. In order to generate the subspace associated with each cluster, we generated random matrices, such that the (i, j)th entry of the matrix was a random real number in the range $(-1, 1)$. In addition, the matrix was made to be symmetric so that the (i, j)th entry was equal to the (j, i)th entry. Since, this is a symmetric matrix, the eigenvalues are all real, and an orthonormal eigenvector system exists. This orthonormal axis system will be used to generate the input orientation for the corresponding cluster. We used a different matrix for each of the clusters in order to generate the corresponding orientation.

The number of points in each cluster was determined by using the following method to determine the number of points in each cluster: We first determined the constant of proportionality r_i which found how many points were present in each cluster i. The constant of proportionality r_i was determined by the formula $r_i = p + R \cdot q$. Here p and q are constants, and R is a random number which is drawn from a uniform distribution in the range $(0, 1)$. (Thus, the resulting number will be drawn from the uniform distribution in the range $(p, p+q)$.) For the purpose of our experiments, we used $p = 1$ and $q = 5$. Thus, if $r_1, r_2, \ldots r_k$ be the proportionality constants for the entire space, then the number of points N_i in the cluster i was determined by using the formula $N_i = N \cdot r_i / \sum_{i=1}^{k} r_i$.

The next step was to generate the anchor points (an approximation of the central point) of each cluster. The anchor points were chosen from a uniform distribution of points in the space. Once the anchor points were determined, they were used to generate the clusters of points. We have already discussed how the axes orientations for each cluster are determined by using the eigenvectors of the randomly generated symmetric matrices. Now it remains to distribute the points in each cluster corresponding to this axis system. Since this axis system corresponds to one in which the correlations of the points are zero, we generate the coordinates with respect to this transformed subspace independently from one another. (The generated points can then be transformed back to their original space.) For each cluster, we picked the l eigenvectors which defined the subspace in which it was hidden. The points were distributed in a uniform random distribution along the other $d - l$ axes directions.

It now remains to explain how the point distribution for the hidden axes directions was accomplished. The first step was to determine the spread along each of the l eigenvectors. This was determined by a pair of spread parameters μ_s and γ. The spread along the eigenvector i was determined to be proportional to $(Q_i)^\gamma$, where Q_i was drawn from an exponential distribution with mean μ_s, and γ was a parameter which determined the level of disproportionality that we wished to create in the various axes directions. This is because when the value of γ is increased, the amount of disproportionality in the values of $(Q_i)^\gamma$ increased for different values of i. Thus, the ith coordinate for the point with respect to the transformed subspace was drawn from a normal distribution with its mean at the anchor point and its variance equal to $(Q_i)^\gamma$. For our experiments, we used $d = 20$, $\mu_s = 0.1$, $\gamma = 2$, $p = 1$, $q = 5$, $k = 5$ and $l = 6$.

The resulting clusters were very sparse in full dimensional space. The cluster sparsity coefficient when measured in full dimensional space for the input clusters for this choice of parameters was 0.85. Furthermore, we checked the sparsity coefficient of *each individual input cluster for each 1-dimensional* [2] *axis parallel projection*, and found that in each case, the cluster sparsity coefficient was at least as high as 0.7, and averaged at 0.83. The high sparsity coefficients indicate that the generated data is one on which either full-dimensional or axis-parallel projected clustering would be meaningless.

3.2 Failure of axis-parallel projections

We tested an axis-parallel version[3] of this algorithm

[2] Cluster sparsity coefficients for higher dimensional projections would always be at least as large as the minimum sparsity coefficient of some 1-dimensional subset, and will be at most as large as the maximum sparsity coefficient of any 1-dimensional subset. This can be proved from the additive property of the energy metric over different dimensions. A formal proof is omitted.

[3] We have another axis-parallel projected clustering algorithm called PROCLUS [1] which also could not match input to output clusters well. In addition, PROCLUS would classify an unusually large number of input points as outliers. We found that the axis-parallel version of ORCLUS was slightly more stable and accurate than PROCLUS.

Input Clusters	A	B	C	D	E
Output Clusters					
1	639	178	0	0	131
2	0	931	549	465	29
3	121	35	851	265	90
4	50	225	135	2889	408
5	135	133	201	128	1412

Table 1: Axis Parallel projections: Confusion Matrix -[User-specified $l = 14$]

Input Clusters	A	B	C	D	E
Output Clusters					
1	367	202	261	208	263
2	70	835	631	1331	241
3	169	101	604	103	73
4	108	180	99	2001	128
5	231	184	141	104	1365

Table 2: Axis Parallel projections: Confusion Matrix -[User-specified $l = 6$]

on the data set. This was done by ensuring that the $FindVectors(\cdot)$ procedure returned the least spread axis parallel directions. We will state the salient observations very briefly:

- The axis parallel projected version found very poor matching between the input and output clusters. The method was not significantly better than a random clustering technique for some of the runs. We have illustrated two runs on the same data set of 10,000 points, with varying values of the projected dimensionality and illustrated the results in Tables 1 and 2. The Confusion Matrix in Table 1 ($l = 14$) is slightly better than the confusion matrix in Table 2 ($l = 6$), though neither of them can be considered very clean partitionings, since each input cluster is split into several output clusters and vice-versa. This was part of an interesting trend that we observed: *reducing the user-specified value of projected dimensionality l always worsened the quality of clustering by the axis-parallel version.* When the clusters exist in arbitrary subspaces, then forcing particular kinds of projections on the clusters leads to loss of information; the greater the projection, the greater the loss.
- The axis-parallel version was unstable. Slight changes in initialization conditions (such as changing the random seed) significantly changed the the matrices obtained in Tables 1 and 2.

The above results are quite expected; our observations on the sparsity coefficients indicate the unsuitability of the class of axis-parallel projection methods.

3.3 Results for ORCLUS

Input Clusters / Output Clusters	A	B	C	D	E
1	898	0	0	0	2
2	0	1401	0	124	0
3	23	0	1703	0	0
4	0	40	0	3623	59
5	24	61	33	0	2009

Table 3: ORCLUS: Confusion Matrix (Case 1) [User-specified $l = 6$]

Input Clust. / Output Clust.	A	B	C	D	E
1	9201	103	12	0	0
2	122	16144	7	24	0
3	0	0	29821	0	22
4	3	0	0	20451	24
5	15	101	0	0	23950

Table 4: ORCLUS: Confusion Matrix (Case 2) [User-specified $l = 6$]

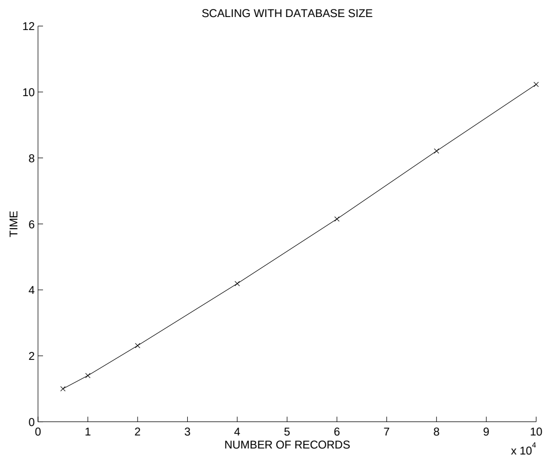

Figure 6: Scaling of running time with database size

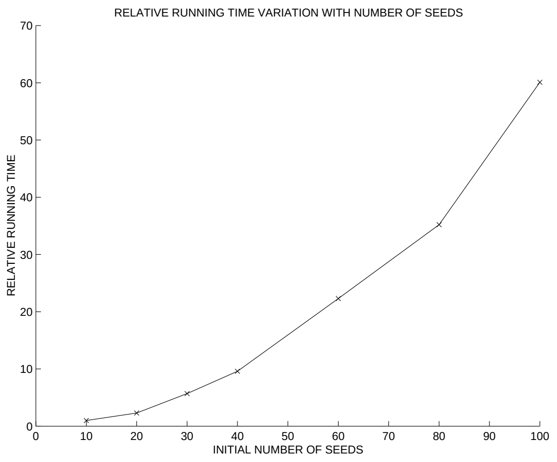

Figure 7: Scaling of running time with k_0

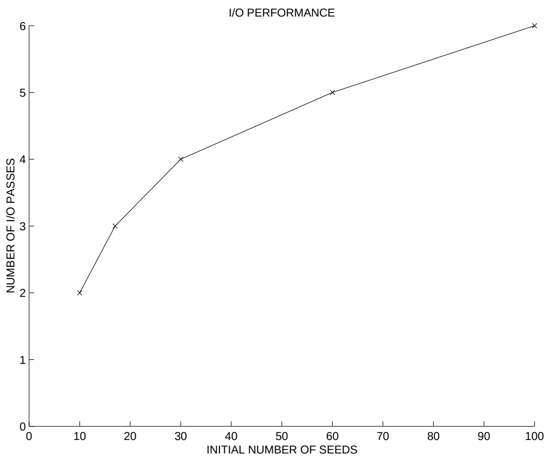

Figure 8: Scaling of I/O passes with k_0

For the purpose of our tests, we always used $k_0 = 15 \cdot k$, unless mentioned otherwise. We generated several input data sets and computed the Confusion Matrix for each of the cases. We will present some of the confusion matrices which are very representative of the general trend that we observed. First we ran the algorithm for different values of the projected dimensionality l and tested the corresponding cluster sparsity coefficient. In each case, we found that the output cluster sparsity coefficient dropped significantly from $l = 7$ to $l = 6$ (Case 1- 0.081 to 0.003; Case 2- 0.09 to 0.0024), whereas the drop from $l = 6$ to $l = 5$ was relatively small (Case 1- 0.003 to 0.0025; Case 2- 0.0024 to 0.0022). In fact, the percentage drop in sparsity coefficient was always the highest from $l = 7$ to $l = 6$ which was the dimensionality of the subspace in which the input clusters were hidden. We used this criterion to pick the projected dimensionality $l = 6$ in both the cases that we illustrate below. The first data set (Case 1) contained 10,000 points, and the results are indicated in Table 3. This is the same data set which was used in Tables 1 and 2. As we can see from the table one of the entries in each column is clearly much larger than the rest of the entries. This indicates that each input cluster gets directed into one output cluster with the exception of some points, which get distributed to other clusters. Table 4 (Case 2) illustrates another example containing 100,000 points. The trends are very similar to Table 3. These results are generally indicative of a very clean mapping from the input to output clusters. We also tested the sensitivity of this clustering to variations in the value of the input parameter l. We found in each case that a good confusion matrix was obtained in the range of $l = 2$ to $l = 8$, and was only slightly worse than the confusion matrix for the optimized value $l = 6$. This is also an indication of the stability of this technique.

We also present the computational scalability results for the algorithm here. The results were averaged over five runs in each case in order to smooth the curve. The scaling of running time with the number of points N in the database is illustrated in Figure 6. As we see, the algorithm performance is practically linear with database size. Note that the curve is interesting only for the case when N is substantially larger than k_0, the initial number of seeds.

Figure 7 illustrates the CPU performance when the initial number of seeds k_0 was increased. We know that the quality of the clusters is likely to be better when starting with a larger number of seeds, because each cluster is then likely to be covered by at least one seed. Correspondingly, the running time of the algorithm also increases, because of the contribution of the subspace determination operations during the *Merge* phase. The I/O requirements varied less dramatically with the initial number of seeds. This is because the total number of iterations is $\log_\alpha(k/k_0)$, in each of which at most one I/O pass is performed. Figure 8 illustrates the corresponding trend. This curve remains the same irrespective of database size, dimensionality, or the final number of projected dimensions. These results indicate that when a larger number of initial seeds are picked in the interest of greater accuracy, the method is unlikely to be I/O bound.

4 Conclusions and Summary

In this paper we discussed the concept of finding arbitrarily oriented projected clusters in high dimensional spaces, a definition of clustering which is a practical and effective solution to the dimensionality curse for the traditional version of this problem. The idea of eliminating the most sparse subspaces for each cluster, and projecting the points into those subspaces in which the greatest similarity occurs is a very generalized notion of clustering of which the full dimensional case is a special one. Since the sparsity of high dimensional data prevents the detection of natural clusters for full-dimensional problems, this modified definition of clustering is best likely to redefine our understanding of the notion of high dimensional clustering. In future research, we will show how to use this technique for effective high dimensional data visualization.

References

[1] C. C. Aggarwal et. al. Fast algorithms for projected clustering. *ACM SIGMOD Conference*, 1999.

[2] R. Agrawal et. al. Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications. *ACM SIGMOD Conference,* 1998.

[3] K. Beyer et. al. When is nearest neighbor meaningful? *ICDT Conference*, 1999.

[4] C. Cheng, A. W. Fu, Y. Zhang. Entropy-based Subspace Clustering for Mining Numerical Data. *ACM SIGKDD Conference*, 1999.

[5] M. Ester et. al. A Density Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. *KDD Conference*, 1996.

[6] C. Faloutsos, K.-I. Lin. FastMap: A Fast Algorithm for Indexing, Data-Mining and Visualization of Traditional and Multimedia Datasets. *ACM SIGMOD Conference*, 1995.

[7] A. Gionis, P. Indyk, R. Motwani. Similarity Search in High Dimensions via Hashing. *VLDB Conference*, 1999.

[8] S. Guha, R. Rastogi, K. Shim. CURE: An Efficient Clustering Algorithm for Large Databases. *ACM SIGMOD Conference*, 1998.

[9] A. Hinneburg, D. Keim. Optimal Grid-Clustering: Towards Breaking the Curse of Dimensionality in High-Dimensional Clustering. *VLDB Conference*, 1999.

[10] P. Indyk, R. Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. *STOC*, 1998.

[11] A. Jain, R. Dubes. *Algorithms for Clustering Data.* Prentice Hall, New Jersey, 1998.

[12] I. T. Jolliffe. *Principal Component Analysis.* Springer-Verlag, New York, 1986.

[13] J. Kleinberg. Two algorithms for nearest-neighbor search in high dimensional space. *STOC*, 1997.

[14] R. Kohavi, D. Sommerfield. Feature Subset Selection Using the Wrapper Method: Overfitting and Dynamic Search Space Topology. *KDD*, 1995.

[15] R. Ng, J. Han. Efficient and Effective Clustering Methods for Spatial Data Mining. *VLDB Conference*, 1994.

[16] K. Ravi Kanth, D. Agrawal, A. Singh. Dimensionality Reduction for Similarity Searching in Dynamic Databases. *ACM SIGMOD Conference*, 1998.

[17] X. Xu et. al. A Distribution-Based Clustering Algorithm for Mining in Large Spatial Databases. *ICDE Conference*, 1998.

[18] T. Zhang, R. Ramakrishnan, M. Livny. BIRCH: An Efficient Data Clustering Method for Very Large Databases. *ACM SIGMOD Conference,* 1996.