Local Dimensionality Reduction: A New Approach to Indexing High Dimensional Spaces *

Kaushik Chakrabarti
Department of Computer Science
University of Illinois
Urbana, IL 61801
kaushikc@cs.uiuc.edu

Sharad Mehrotra
Department of Information and Computer Science
University of California
Irvine, CA 92697
sharad@ics.uci.edu

Abstract

Many emerging application domains require database systems to support efficient access over highly multidimensional datasets. The current state-of-the-art technique to indexing high dimensional data is to first reduce the dimensionality of the data using Principal Component Analysis and then indexing the reduced-dimensionality space using a multidimensional index structure. The above technique, referred to as global dimensionality reduction (GDR), works well when the data set is globally correlated, i.e. most of the variation in the data can be captured by a few dimensions. In practice, datasets are often not globally correlated. In such cases, reducing the data dimensionality using GDR causes significant loss of distance information resulting in a large number of false positives and hence a high query cost. Even when a global correlation does not exist, there may exist subsets of data that are locally correlated. In this paper, we propose a technique called Local Dimensionality Reduction (LDR) that tries to find local correlations in the data and performs dimensionality reduction on the locally correlated clusters of data individually. We develop an index structure that exploits the correlated clusters to efficiently support point, range and k-nearest neighbor queries over high dimensional datasets. Our experiments on synthetic as well as real-life datasets show that our technique (1) reduces the dimensionality of the data with significantly lower loss in distance information compared to GDR and (2) significantly outperforms the GDR, original space indexing and linear scan techniques in terms of the query cost for both synthetic and real-life datasets.

* This work was supported by NSF CAREER award IIS-9734300, and in part by the Army Research Laboratory under Cooperative Agreement No. DAAL01-96-2-0003.

**Proceedings of the 26th VLDB Conference,
Cairo, Egypt, 2000.**

1 Introduction

With an increasing number of new database applications dealing with highly multidimensional datasets, techniques to support efficient query processing over such data sets has emerged as an important research area. These applications include multimedia content-based retrieval, exploratory data analysis/data mining, scientific databases, medical applications and time-series matching. To provide efficient access over high dimensional feature spaces (HDFS), many indexing techniques have been proposed in the literature. One class of techniques comprises of *high dimensional index trees* [3, 15, 16, 5]. Although these index structures work well in low to medium dimensionality spaces (upto 20-30 dimensions), a simple sequential scan usually performs better at higher dimensionalities [4, 20].

To scale to higher dimensionalities, a commonly used approach is *dimensionality reduction* [9]. This technique has been proposed for both multimedia retrieval and data mining applications. The idea is to first reduce the dimensionality of the data and then index the reduced space using a multidimensional index structure [7]. Most of the information in the dataset is condensed to a few dimensions (the first few principal components (PCs)) by using principal component analysis (PCA). The PCs can be arbitrarily oriented with respect to the original axes [9]. The remaining dimensions (i.e. the later components) are eliminated and the index is built on the reduced space. To answer queries, the query is first mapped to the reduced space and then executed on the index structure. Since the distance in the reduced-dimensional space lower bounds the distance in the original space, the query processing algorithm can guarantee no false dismissals [7]. The answer set returned can have false positives (i.e. false admissions) which are eliminated before it is returned to the user. We refer to this technique as *global dimensionality reduction* (GDR) i.e. dimensionality reduction over the *entire* dataset taken together.

GDR works well when the dataset is *globally correlated* i.e. most of the variation in the data can be captured by a few orthonormal dimensions (the first few PCs). Such a case is illustrated in Figure 1(a) where a single dimension (the first PC) captures the variation of data in the 2-d space. In such cases, it is possible to eliminate most of the dimensions (the later PCs) with little or no loss of distance information. However, in practice, the dataset may not be globally correlated (see Figure 1(b)). In such cases, reducing the data dimensionality using GDR will cause a significant loss of distance information. Loss in distance information is manifested by a large number of false positives

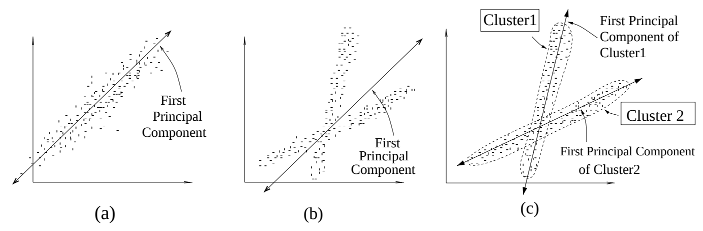
First Principal Component
(a)
First Principal Component
(b)
Cluster1
First Principal Component of Cluster1
Cluster 2
First Principal Component of Cluster2
(c)

Figure 1: Global and Local Dimensionality Reduction Techniques (a) GDR(from 2-d to 1-d) on globally correlated data (b) GDR (from 2-d to 1-d) on globally non-correlated (but locally correlated) data (c) LDR (from 2-d to 1-d) on the same data as in (b)

and is measured by precision [14] (cf. Section 5). More the loss, larger the number of false positives, lower the precision. False positives increase the cost of the query by (1) causing the query to make unnecessary accesses to nodes of the index structure and (2) adding to the post-processing cost of the query, that of checking the objects returned by the index and eliminating the false positives. The cost increases with the increase in the number of false positives. Note that false positives do not affect the quality the answers as they are not returned to the user.

Even when a global correlation does not exist, there may exist subsets of data that are *locally correlated* (e.g., the data in Figure 1(b) is not globally correlated but is locally correlated as shown in Figure 1(c)). Obviously, the correlation structure (the PCs) differ from one subset to another as otherwise they would be globally correlated. We refer to these subsets as *correlated clusters* or simply *clusters*. [1] In such cases, GDR would not be able to obtain a single reduced space of desired dimensionality for the entire dataset without significant loss of query accuracy. If we perform dimensionality reduction on each cluster *individually* (assuming we can find the clusters) rather than on the entire dataset, we can obtain a set of different reduced spaces of desired dimensionality (as shown in Figure 1(c)) which together cover the entire dataset [2] but achieves it with minimal loss of query precision and hence significantly lower query cost. We refer to this approach as local dimensionality reduction (LDR).

Contributions: In this paper, we propose LDR as an approach to high dimensional indexing. Our contributions can be summarized as follows:

- We develop an algorithm to discover correlated clusters in the dataset. Like any clustering problem, the problem, in general, is NP-Hard. Hence, our algorithm is heuristic-based. Our algorithm performs dimensionality reduction of each cluster individually to obtain the reduced space (referred to as subspace) for each cluster. The data items that do not belong to any cluster are outputted as outliers. The algorithm allows the user to control the amount of information loss incurred by dimensionality reduction and hence the query precision/cost.
- We present a technique to index the subspaces individually. We present query processing algorithms for point, range and k-nearest neighbor (k-NN) queries that execute on the index structure. Unlike many previous techniques [14, 19], our algorithms guarantee correctness of the result i.e. returns exactly the same answers as if the query executed on the original space. In other words, the answer set returned to the user has no false positives or false negatives.
- We perform extensive experiments on synthetic as well as real-life datasets to evaluate the effectiveness of LDR as an indexing technique and compare it with other techniques, namely, GDR, index structure on the original HDFS (referred to as the original space indexing (OSI) technique) and linear scan. Our experiments show that (1) LDR can reduce dimensionality with significantly lower loss in query precision as compared to GDR technique. For the same reduced dimensionality, LDR outperforms GDR by almost an order of magnitude in terms of precision. and (2) LDR performs significantly better than other techniques, namely GDR, original space indexing and sequential scan, in terms of query cost for both synthetic and real-life datasets.

Roadmap: The rest of the paper is organized as follows. In Section 2, we provide an overview of related work. In Section 3, we present the algorithm to discover the correlated clusters in the data. Section 4 discusses techniques to index the subspaces and support similarity queries on top of the index structure. In Section 5, we present the performance results. Section 6 offers the final concluding remarks.

2 Related Work

Previous work on high dimensional indexing techniques includes development of high dimensional index structures (e.g., X-tree[3], SR-tree [15], TV-tree [16], Hybrid-tree [5]) and global dimensionality reduction techniques [9, 7, 8, 14]. The techniques proposed in this paper build on the above work. Our work is also related to the clustering algorithms that have been developed recently for database mining (e.g., BIRCH, CLARANS, CURE algorithms) [21, 18, 12]. The algorithms most related to this paper are those that discover patterns in low dimensional subspaces [1, 2]. In [1], Agarwal et. al. present an algorithm, called CLIQUE, to discover“dense” regions in all subspaces of the original data space. The algorithm works from lower to higher dimensionality subspaces: it starts by discovering 1-d dense units and iteratively discovers all dense units in each k-d subspace by building from the dense units in (k-1)-d subspaces. In [2], Aggarwal et. al. present an algorithm, called PROCLUS, that clusters the data based on their correlation i.e. partitions the data into disjoint groups of correlated points. The authors use the hill climbing technique, popular in spatial cluster analysis, to determine the projected clusters. Neither CLIQUE, nor PROCLUS can be used as an LDR technique since they cannot discover clusters when the principal components are arbitrarily oriented. They can discover only those clusters that are correlated along one or more of the original dimensions. The above techniques are meant for discovering interesting patterns in the data; since correlation along arbitrarily oriented components is usually not that interesting to the user, they do not attempt to discover such correlation. On the contrary, the goal of LDR is efficient indexing; it must be able to discover such correlation in order to minimize the loss of information and make indexing efficient. Also, since the motivation of their work is pattern discovery and not indexing, they do not address the indexing and query processing issues which we have addressed in this paper. To the best of our knowledge, this is the first paper that pro-

[1]Note that correlated clusters (formally defined in Section 3) differ from the usual definition of clusters i.e. a set of spatially close points. To avoid confusion, we refer to the latter as *spatial clusters* in this paper.

[2]The set of reduced spaces may not necessarily cover the entire dataset as there may be outliers. We account for outliers in our algorithm.

Symbols	Definitions
N	Number of objects in the database
M	Maximum number of clusters desired
K	Actual number of clusters found ($K \leq M$)
D	Dimensionality of the original feature space
S_i	The ith cluster
C_i	Centroid of S_i
n_i	Size of S_i (number of objects)
$\mathcal{A}_i$	Set of points in S_i
Φ_i	The principal components of S_i
$\Phi_i^{(j)}$	The jth principal component of S_i
d_i	Subspace dimensionality of S_i
ϵ	Neighborhood range
$MaxReconDist$	Maximum Reconstruction distance
$FracOutliers$	Permissible fraction of outliers
$MinSize$	Minimum Size of a cluster
$MaxDim$	Maximum subspace dimensionality of a cluster
$\mathcal{O}$	Set of outliers

Table 1: Summary of symbols and definitions

poses to exploit the local correlations in data for the purpose of indexing.

3 Identifying Correlated Clusters

In this section, we formally define the notion of correlated clusters and present an algorithm to discover such clusters in the data.

3.1 Definitions

In developing the algorithm to identify the correlated clusters, we will need the following definitions.

Definition 1 (Cluster and Subspace) Given a set $\mathcal{A}$ of N points in a D-dimensional feature space, we define a *cluster* S as a set $\mathcal{A}_S$ ($\mathcal{A}_S \subseteq \mathcal{A}$) of locally correlated points. Each cluster S is defined by $S = \langle \Phi_S, d_S, C_S, \mathcal{A}_S \rangle$ where:

- Φ_S are the principal components of the cluster, $\Phi_S^{(i)}$ denoting the ith principal component.
- d_S is the reduced dimensionality i.e. the number of dimensions retained. Obviously, the retained dimensions correspond to the first d_S principal components $\Phi_S^{(i)}, 1 \leq i \leq d_S$ while the eliminated dimensions correspond to the next $(D - d_S)$ components. Hence we use the terms (principal) components and dimensions interchangeably in the context of the transformed space.
- $C_S = [C_S^{(d_S+1)} \cdots C_S^{(D)}]$ is the centroid, that stores, for each eliminated dimension $\Phi_i, (d_S + 1) \leq i \leq D$, a single constant which is "representative" of the position of every point in the cluster along this unrepresented dimension (as we are not storing their unique positions along these dimensions).
- $\mathcal{A}_S$ is the set of points in the cluster

The reduced dimensionality space defined by $\Phi_S^{(i)}, 1 \leq i \leq d_S$ is called the *subspace* of S. d_S is called the subspace dimensionality of S.

■

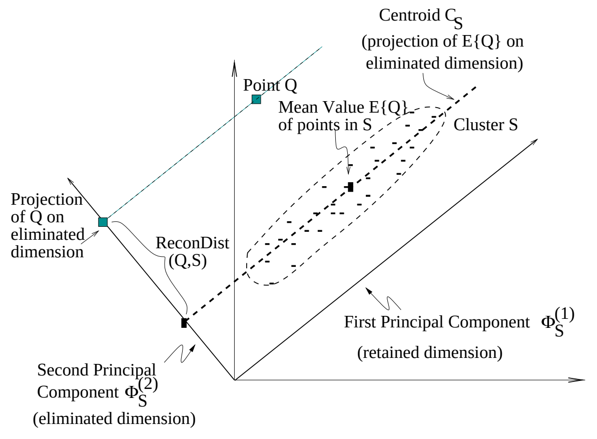
Centroid C_S
(projection of E{Q} on eliminated dimension)
Point Q
Mean Value E{Q} of points in S
Cluster S
Projection of Q on eliminated dimension
ReconDist (Q,S)
First Principal Component $\Phi_S^{(1)}$
(retained dimension)
Second Principal Component $\Phi_S^{(2)}$
(eliminated dimension)

Figure 2: Centroid and Reconstruction Distance.

Definition 2 (Reconstruction Vector) Given a cluster $S = \langle \Phi_S, d_S, C_S, \mathcal{A}_S \rangle$, we define the *reconstruction vector* $\overline{ReconVect}(Q, S)$ of a point Q from S as follows:

$$\overline{ReconVect}(Q, S) = \bar{\Sigma}_{i=(d_S+1)}^{D} (Q \bullet \Phi_S^{(i)} - C_S^{(i)}) \Phi_S^{(i)} \quad (1)$$

where $\bar{\Sigma}$ denotes vector addition and $\bullet$ denotes scalar product (i.e. $Q \bullet \Phi_S^{(i)}$ is the projection of Q on $\Phi_S^{(i)}$ as shown in Figure 2). $(Q \bullet \Phi_S^{(i)} - C_S^{(i)})$ is the (scalar) distance of Q from the centroid along each eliminated dimension and $\overline{ReconVector}(Q, S)$ is the vector of these distances.

■

Definition 3 (Reconstruction Distance) Given a cluster $S = \langle \Phi_S, d_S, C_S, \mathcal{A}_S \rangle$, we now define the *reconstruction distance* (scalar) $ReconDist(Q, S, \mathcal{D})$ of a point Q from S. $\mathcal{D}$ is the distance function used to define the similarity between points in the HDFS. Let $\mathcal{D}$ be an L_p metric i.e. $\mathcal{D}(P, P') = \| P - P' \|_p = [\Sigma_{i=1}^{d} (|P[i] - P'[i]|)^p]^{1/p}$. We define $ReconDist(Q, S, \mathcal{D})$ [3] as follows:

$$\begin{aligned} ReconDist(Q, S, \mathcal{D}) &= ReconDist(Q, S, L_p) && (2) \\ &= \| \overline{ReconVect}(Q, S) \|_p && (3) \\ &= [\Sigma_{i=d_S+1}^{D} (|Q \bullet \Phi_S^{(i)} - C_S^{(i)}|)^p]^{1/p} && (4) \end{aligned}$$

■

Note that for any point Q mapped to the d_S-dimensional subspace of S, $\overline{ReconVect}(Q, S)$ (and $ReconDist(Q, S)$) represent the error in the representation i.e. the vector (and scalar) distance between the exact D-dimensional representation of Q and its approximate representation in the d_S-dimensional subspace of S. Higher the error, more the amount of distance information lost.

3.2 Constraints on Correlated Clusters

Our objective in defining clusters is to identify low dimensional subspaces, one for each cluster, that can be indexed separately.

[3] Assuming that $\mathcal{D}$ is a fixed L_p metric, we usually omit the $\mathcal{D}$ in $ReconDist(Q, S, \mathcal{D})$ for simplicity of notation.

We desire each subspace to have as low dimensionality as possible without losing too much distance information. In order to achieve the desired goal, each cluster must satisfy the following constraints:

1. **Reconstruction Distance Bound:** In order to restrict the maximum representation error of any point in the low dimensional subspace, we enforce the reconstruction distance of any point $P \in \mathcal{A}_S$ to satisfy the following condition: $ReconDist(P,S) \leq MaxReconDist$ where $MaxReconDist$ is a parameter specified by the user. This condition restricts the amount of information lost within each cluster and hence guarantees a high precision which in turn implies lower query cost.

2. **Dimensionality Bound:** For efficient indexing, we want the subspace dimensionality to be as low as possible while still maintaining high query precision. A cluster must not retain any more dimensions that necessary. In other words, it must retain the minimum number of dimensions required to accommodate the points in the dataset. Note than a cluster S can accommodate a point P only if $ReconDist(P,S) \leq MaxReconDist$. To ensure that the subspace dimensionality d_S is below the critical dimensionality of the multidimensional index structure (i.e. the dimensionality above which a sequential scan is better), we enforce the following condition: $d_S \leq MaxDim$ where $MaxDim$ is specified by the user.

3. **Choice of Centroid:** For each cluster S, we use PCA to determine the subspace i.e. Φ_S is the set of eigenvectors of the covariance matrix of $\mathcal{A}_S$ sorted based on their eigenvalues. [9] shows that for a given choice of reduced dimensionality d_S, the representation error is minimized by choosing the first d_S components among Φ_S and choosing C_S to be the mean value of the points (i.e. the centroid) projected on the eliminated dimensions. To minimize the information loss, we choose $C_S^{(i)} = E\{P \bullet \Phi_S^{(i)}\} = E\{P\} \bullet \Phi_S^{(i)}$ (see Figure 2).

4. **Size Bound:** Finally, we desire each cluster to have a minimum cardinality (number of points) : $n_S \geq MinSize$ where $MinSize$ is user-specified. The clusters that are too small are considered to be outliers.

The goal of the LDR algorithm described below is to discover the set $\mathcal{S} = S_1, S_2, ..., S_K$ of K clusters (where $K \leq M$, M being the maximum number of clusters desired) that exists in the data and that satisfy the above constraints. The remaining points, that do not belong to any of the clusters, are placed in the outlier set $\mathcal{O}$.

3.3 The Clustering Algorithm

Since the LDR algorithm needs to perform *local* correlation analysis (i.e. PCA on subsets of points in the dataset rather than the whole dataset), we need to first identify the right subsets to perform the analysis on. This poses a cyclic problem: how do we identify the right subsets without doing the correlation analysis and how do we do the analysis without knowing the subsets. We break the cycle by using *spatial clusters* as an initial guess of the right subsets. Then we perform PCA on each spatial cluster individually. Finally, we 'recluster' the points based on the correlation information (i.e. principal components) to obtain the correlated clusters. The clustering algorithm is shown in Table 2. It takes a set of points $\mathcal{A}$ and a set of clusters $\mathcal{S}$ as input. When it is invoked for the first time, $\mathcal{A}$ is the entire dataset and each cluster in $\mathcal{S}$ is marked 'empty'. At the end, each identified cluster is marked 'complete' indicating a completely constructed cluster (no further change); the remaining clusters remain marked 'empty'. The points that do not belong to any of the clusters are placed to the outlier set $\mathcal{O}$. The details of each step is described below:

```
Clustering Algorithm
Input: Set of Points $\mathcal{A}$, Set of clusters $\mathcal{S}$ (each cluster is either empty or complete)
Output: Some empty clusters are completed, the remaining points form the set of outliers $\mathcal{O}$
FindClusters($\mathcal{A}, \mathcal{S}, \mathcal{O}$)

FC1: For each empty cluster, select a random point $P \in \mathcal{A}$ such that $P$ is sufficiently far from all completed and valid clusters. If found, make $P$ the centroid $C_i$ and mark $S_i$ valid.

FC2: For each point $P \in \mathcal{A}$, add $P$ to the closest valid cluster $S_i$ (i.e. $i = argmin(Distance(P, C_i))$) if $P$ lies in the $\epsilon$-neighborhood of $C_i$ i.e. $Distance(P, C_i) \leq \epsilon$.

FC3: For each valid cluster $S_i$, compute the principal components $\Phi_i$ using PCA. Remove all points from $\mathcal{A}_i$.

FC4: For each point $P \in \mathcal{A}$, find the valid cluster $S_i$ that, among all the valid clusters requires the minimum subspace dimensionality $LD(P)$ to satisfy $ReconDist(P, S_i) \leq MaxReconDist$ (break ties arbitrarily). If $LD(P) \leq MaxDim$, increment $V_i[j]$ for $j = 0$ to $(LD(P) - 1)$ and $n_i$.

FC5: For each valid cluster $S_i$, compute the subspace dimensionality $d_i$ as: $d_i = \{j | F_i[j] \leq FracOutliers$ and $F_i[j-1] > FracOutliers\}$ where $F_i[j] = \frac{V_i[j]}{n_i}$.

FC6: For each point $P \in \mathcal{A}$, add $P$ to the first valid cluster $S_i$ such that $ReconDist(P, S_i) \leq MaxReconDist$. If no such $S_i$ exists, add P to $\mathcal{O}$.

FC7: If a valid cluster $S_i$ violates the size constraint i.e. ($|\mathcal{A}_i| < MinSize$), mark it empty. Remove each point $P \in \mathcal{A}_i$ from $S_i$ and add it to the first succeeding cluster $S_j$ that satisfies $ReconDist(P, S_j) \leq MaxReconDist$ or to $\mathcal{O}$ if there is no such cluster. Mark the other valid clusters complete. For each complete cluster $S_i$, map each point $P \in \mathcal{A}_i$ to the subspace and store it along with $ReconDist(P, S, \mathcal{D})$.
```

Table 2: Clustering Algorithm

- **Construct Spatial Clusters**(Steps FC1 and FC2): The algorithm starts by constructing M spatial clusters where M is the maximum number of clusters desired. We use a simple single-pass partitioning-based spatial clustering algorithm to determine the spatial clusters [18]. We first choose a set of $\mathcal{C} \subset \mathcal{A}$ of *well-scattered* points as the centroids such that points that belong to the same spatial cluster are not chosen to serve as centroids to different clusters. Such a set $\mathcal{C}$ is called a *piercing* set [2]. We achieve

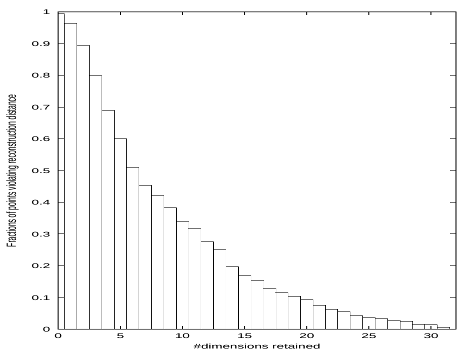

Figure 3: Determining subspace dimensionality (MaxDim=32).

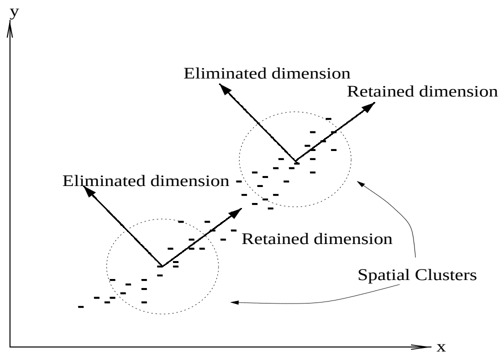

Figure 4: Splitting of correlated clusters due to initial spatial clustering.

this by ensuring that each point $P \in \mathcal{C}$ in the set is sufficiently far from any already chosen point $P' \in \mathcal{C}$ i.e. $Dist(P, P') > threshold$ for a user-defined threshold. [4] This technique, proposed by Gonzalez [10], is guaranteed to return a piercing if no outliers are present. To avoid scanning though the whole database to choose the centroids, we first construct a random sample of the dataset and choose the centroids from the sample [2, 12]. We choose the sample to be large enough (using Chernoff bounds [17]) such that the probability of missing clusters due to sampling is low i.e. there is at least one point from each cluster present in the sample with a high probability [12]. Once the centroids are chosen, we group each point $P \in \mathcal{A}$ with the closest centroid $C_{closest}$ if $Distance(P, C_{closest}) \leq \epsilon$ and update the centroid to reflect the mean position of its group. If $Distance(P, C_{closest}) > \epsilon$, we ignore P. The restriction of the neighborhood range to ϵ makes the correlation analysis *localized*. Smaller the value of ϵ, the more localized the analysis. At the same time, ϵ has to be large enough so that we get a sufficiently large number of points in the cluster which is necessary for the correlation analysis to be robust.

- **Compute PCs**(Step FC3): Once we have the spatial clusters, we perform PCA on each spatial cluster S_i individually to obtain the principal components $\Phi_S^{(i)}, i = [1, D]$. We do not eliminate any components yet. We compute the mean value M_i of the points in S_i so that we can compute $ReconDist(P, S_i)$ in Steps FC4 and FC5 for any choice of subspace dimensionality d_i. Finally, we remove the points from the spatial clusters so that they can be reclustered as described in Step FC6.

- **Determine Subspace Dimensionality**(Steps FC4 and FC5): For each cluster S_i, we must retain no more dimensions than necessary to accommodate the points in the dataset (except the outliers). To determine the number of dimensions d_i to be retained for each cluster S_i, we first determine, for each point $P \in \mathcal{A}$, the best cluster, if one exists, for placing P. Let $LD(P, S_i)$ denote the the least dimensionality needed for the cluster S_i to represent P with $ReconDist(P, S_i) \leq MaxReconDist$. Formally,

$$\begin{aligned} LD(P, S_i) = \{d| \\ ReconDist(P, S_i) \leq MaxReconDist \text{ if } d_i \geq d \\ \text{and } ReconDist(P, S_i) > MaxReconDist \text{ otherwise } \} \end{aligned} \quad (5)$$

In other words, the first $LD(P, S_i)$ PCs are just enough to satisfy the above constraint. Note that such a $LD(P, S_i)$ always exists for a non-negative $MaxReconDist$. Let $LD(P) = min\ \{\ LD(P, S_i)|S_i$ is a valid cluster $\}$. If $LD(P) \leq MaxDim$, there exists a cluster that can accommodate P without violating the dimensionality bound. Let $LD(P, S_i) = LD(P)$ (if there are multiple such clusters S_i, break ties arbitrarily). We say S_i is the "best" cluster for placing P since S_i is the cluster that, among all the valid clusters, needs to retain the minimum number of dimensions to accommodate P. P would satisfy the $ReconDist(P, S_i) \leq MaxReconDist$ bound if the subspace dimensionality d_i of S_i is such that $LD(P, S_i) \leq d_i \leq MaxDim$ and would violate it if $0 \leq d_i < LD(P, S_i)$. For each cluster S_i, we maintain this information as a count array $V_i[j], j = [0, MaxDim]$ where $V_i[j]$ is the number of points that, among the points chosen to be placed in S_i, would violate the $ReconDist(P, S_i) \leq MaxReconDist$ constraint if the subspace dimensionality d_i is j: so in this case (for point P), we must increment $V_i[j]$ for $j = 0$ to $(LD(P, S_i) - 1)$ and the total count n_i of points chosen to be placed in S_i. ($V_i[j]$ and n_i is initialized to 0 before FC4 begins). On the other hand, if $LD(P) > MaxDim$, there exists no cluster in which P can be placed without violating the dimensionality bound; so we do nothing.

At the end of the pass over the dataset, for each cluster S_i, we have computed $V_i[j], j = [0, MaxDim]$ and n_i. We use this to compute $F_i[j], j = [0, MaxDim]$ where $F_i[j]$ is the fraction of points that, among those chosen to be placed in S_i (during FC4), would violate the $ReconDist(P, S_i) \leq MaxReconDist$ constraint if the subspace dimensionality d_i is j i.e. $F_i[j] = \frac{V_i[j]}{n_i}$. An example of F_i from one of the experiments conducted on the real life dataset (cf. Section 5.3) is shown in Figure 3. We choose d_i to be as low as possible without too many points violating the reconstruction distance bound i.e. not more than $FracOutliers$ fraction of points in S_i where $FracOutliers$ is specified by the

[4]For subsequent invocations of FindClusters procedure during the iterative algorithm (Step 2 in Table 3), there may exist already completed clusters (does not exist during the initial invocation). Hence P must also be sufficiently far from all complete clusters formed so far i.e. $ReconDist(P, S) > threshold$ for each complete cluster S.

user. In other words, d_i is the minimum number of dimensions that must be retained so that the fraction of points that violate the $ReconDist(P, S_i) \leq MaxReconDist$ constraint is no more that $FracOutliers$ i.e. $d_i = \{j | F_i[j] \leq FracOutliers$ and $F_i[j-1] > FracOutliers\}$. In Figure 3, d_i is 21 for $FracOutliers = 0.1$, 16 for $FracOutliers = 0.2$ and 14 for $FracOutliers = 0.3$. We now have all the subspaces formed. In the next step, we assign the points to the clusters.

- **Recluster Points**(Step FC6): In the reclustering step, we reassign each point $P \in \mathcal{A}$ to a cluster S that covers P i.e. $ReconDist(P, S) \leq MaxReconDist$. If there exists no such cluster, P is added to the outlier set $\mathcal{O}$. If there exists just one cluster that covers P, P is assigned to that cluster. Now we consider the interesting case of multiple clusters covering P. In this case, there is a possibility that some of these clusters are actually parts of the same correlated cluster but has been split due to the initial spatial clustering. This is illustrated in Figure 4. Since points in a correlated cluster can be spatially distant from each other (e.g., form an elongated cluster in Figure 4) and spatial clustering only clusters spatially close points, it may end up putting correlated points in different spatial clusters, thus breaking up a single correlated cluster into two or more clusters. Although such 'splitting' does not affect the indexing cost of our technique for range queries and k-NN queries, it increases the cost of point search and deletion as multiple clusters may need to searched in contrast to just one when there is no 'splitting'. (cf. Section 4.2.1). Hence, we must detect these 'broken' clusters and merge them back together. We achieve this by maintaining the clusters in some fixed order (e.g., order in which they were created). For each point $P \in \mathcal{P}$, we check each cluster sequentially in that order and assign it to the first cluster that covers P. If two (or more) clusters are part of the same correlated cluster, most points will be covered by all of them but will *always* be assigned to only one them, whichever appears first in the order. This effectively merges the clusters into one since only the first one will remain while the others will end up being almost empty and will be discarded due to the violation of size bound in FC7. Note that the $FracOutliers$ bound in Step FC5 still holds i.e. besides the points for which $LD(P) > MaxDim$, no more that $FracOutliers$ fraction of points can become outliers.

- **Map Points**(Step FC7): In the final step of the algorithm, we eliminate clusters that violate the size constraint. We remove each point from these clusters and add it to the first succeeding valid cluster S_j that satisfies the $ReconDist(P, S_j) \leq MaxReconDist$ bound or to $\mathcal{O}$ otherwise. For the remaining clusters S_i, we map each point $P \in \mathcal{A}_i$ to the subspace by projecting P to $\Phi_i^{(j)}, 1 \leq j \leq d_i$ and refer it as the (d_i-d) image $Image(P, S_i)$ of P:

$$Image(P, S_i)[j] = P \bullet \Phi_i^{(j)} \text{ for } 1 \leq j \leq d_i \qquad (6)$$

We refer to P as the (D-d) original $Original(Image(P, S_i), S_i)$ of its image $Image(P, S_i)$. We store the image of each point along with the reconstruction distance $ReconDist(P, S_i)$.

Since FindClusters chooses the initial centroids from a random sample, there is a risk of missing out some clusters. One way to reduce this risk is to choose a large number of initial centroids but at the cost of slowing down the clustering algorithm. We reduce the risk of missing clusters by trying to discover more clusters, if there exists, among the points returned as outliers by the initial invocation of FindClusters. We iterate the above process as long as new clusters are still being discovered as shown below:

Iterative Clustering	
(1)	FindClusters($\mathcal{A}$, $\mathcal{S}$, $\mathcal{O}$); /* initial invocation */
(2)	Let $\mathcal{O}'$ be an empty set. Invoke FindClusters($\mathcal{O}$, $\mathcal{S}$, $\mathcal{O}'$). Make $\mathcal{O}'$ the new outlier set i.e. $\mathcal{O} \leftarrow \mathcal{O}'$. If new clusters found, go to (2). Else return.

Table 3: Iterative Clustering Algorithm

The above iterative clustering algorithm is somewhat similar to the hill climbing technique, commonly used in spatial clustering algorithms (especially in partitioning-based clustering algorithms like k-means, k-medoids and CLARANS [18]). In this technique, the "bad quality" clusters (the ones that violate the size bound) are discarded (Step FC7) and is replaced, if possible, by better quality clusters. However, unlike the hill climbing approach where all the points are reassigned to the clusters, we do not reassign the points already assigned to the 'complete' clusters. Alternatively, we can follow the hill climbing approach but it is computationally more expensive and requires more scans of the database [18].

Cost Analysis: The above algorithm requires three passes through the dataset (FC2, FC4 and FC6) and a time complexity of $O(ND^2K)$. The detailed analysis can be found in [6].

4 Indexing Correlated Clusters

Having developed the technique to find the correlated clusters, we now shift our attention to how to use them for indexing. Our objective is to develop a data structure that exploits the correlated clusters to efficiently support range and k-NN queries over HDFSs. The developed data structure must also be able to handle insertions and deletions.

4.1 Data Structure

The data structure, referred to as the global index structure (GI) (i.e. index on entire dataset), consists of separate multidimensional indices for each cluster, connected to a single root node. The global index structure is shown in Figure 5. We explain the various components in details below:

- *The Root Node* R of GI contains the following information for each cluster S_i: (1) a pointer to the root node R_i (i.e. the address of disk block containing R_i) of the cluster index I_i (the multidimensional index on S_i), (2) the principal components Φ_i (3) the subspace dimensionality d_i and (4) the centroid C_i. It also contains an access pointer O to the outlier cluster $\mathcal{O}$. If there is an index on $\mathcal{O}$ (discussed later), O points to the root node of that index; otherwise, it points to the start of the set of blocks on which the outlier set resides on disk. R may occupy one or more disk blocks depending on the number of clusters K and original dimensionality D.

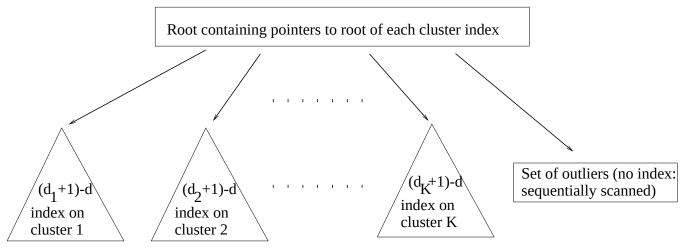
Root containing pointers to root of each cluster index
(d1+1)-d index on cluster 1
(d2+1)-d index on cluster 2
(dK+1)-d index on cluster K
Set of outliers (no index: sequentially scanned)

Figure 5: The global index structure

- *The Cluster Indices:* We maintain a multidimensional index I_i for each cluster S_i in which we store the reduced dimensional representation of the points in S_i. However, instead of building the index I_i on the d_i-d subspace of S_i defined by $\Phi_i^{(j)}, 1 \leq j \leq d_i$, we build I_i on the (d_i+1)-d space, the first d_i dimensions of which are defined by $\Phi_i^{(j)}, 1 \leq j \leq d_i$ as above while the (d_i+1)th dimension is defined by the reconstruction distance $ReconDist(P, S_i, \mathcal{D})$. Including reconstruction distance as a dimension helps to improve query precision (as explained later). We redefine the image $NewImage(P, S_i)$ of a point $P \in \mathcal{A}_i$ as a (d_i+1)-d point (rather than a d_i-d point), incorporating the reconstruction distance as the (d_i+1)th dimension:

$$\begin{aligned} &NewImage(P, S_i)[j] \\ &\quad = Image(P, S_i)[j] = P \bullet \Phi_i^{(j)} \text{ for } 1 \leq j \leq d_i \\ &\quad = ReconDist(P, S_i, \mathcal{D}) \text{ for } j = d_i + 1 \qquad (7) \end{aligned}$$

  The (d_i+1)-d cluster index I_i is constructed by inserting the (d_i+1)-d images (i.e. $NewImage(P, S_i)$) of each point $P \in \mathcal{A}_i$ into the multidimensional index structure using the insertion algorithm of the index structure. Any disk-based multidimensional index structure (e.g., R-tree [13], X-tree [3], SR-tree [15], Hybrid Tree [5]) can be used for this purpose. We used the hybrid tree in our experiments since it is a space partitioning index structure (i.e. has "dimensionality-independent" fanout), is more scalable to high dimensionalities in terms of query cost and can support arbitrary distance metrics [5].
- *The Outlier Index:* For the outlier set $\mathcal{O}$, we may or may not build an index depending on whether the original dimensionality D is below or above the critical dimensionality. In this paper, we assume that D is above the critical dimensionality of the index structure and hence choose not to index the outlier set (i.e. use sequential scan for it).

Like other database index trees (e.g., B-tree, R-tree), the global index (GI) shown in Figure 5 is disk-based. But it may not be perfectly height balanced i.e. all paths from R to leaf may not be of exactly equal length. The reason is that the sizes and the dimensionalities may differ from one cluster to another causing the cluster indices to have different heights. We found that GI is *almost* height balanced (i.e. the difference in the lengths of *any* two paths from R to leaf is never more than 1 or 2) due to the size bound on the clusters (see [6] for details). Also, its height cannot exceed the height of the original space index by more than 1 (see [6] for details).

To guarantee the correctness of our query algorithms (i.e. to ensure no false dismissals), we need to show that the cluster index distances *lower bounds* the actual distances in the original D-d space [7]. In other words, for any two D-d points P and Q, $\mathcal{D}(NewImage(\text{P},S_i), NewImage(\text{Q},S_i))$ must always lower bound $\mathcal{D}(P,Q)$.

Lemma 1 (Lower Bounding Lemma)
$\mathcal{D}(NewImage(P,S_i), NewImage(Q,S_i))$ always lower bounds $\mathcal{D}(P,Q)$. (Proof in [6]).

Note that instead of incorporating reconstruction distance as the (d_i+1)th dimension, we could have simply constructed GI with each cluster index I_i defined on the corresponding d_i-d subspace $\Phi_i^{(j)}, 1 \leq j \leq d_i$. Since the lower bounding lemma holds for the d_i-d subspaces (as shown in [7]), the query processing algorithms described below would have been correct. The reason we use (d_i+1)-d subspace is that the distances in the (d_i+1)-d subspace upper bounds the distances in the d_i-d subspace and hence provides a tighter lower bound to distances in the original D-d space:

$$\begin{aligned} &\mathcal{D}(NewImage(P,S_i), NewImage(Q,S_i)) = \\ &\quad [\mathcal{D}(Image(P,S_i), Image(Q,S_i))^p + \\ &\quad |(ReconDist(P,S_i,\mathcal{D}) - ReconDist(Q,S_i,\mathcal{D}))|^p]^{1/p} \\ &\quad \Rightarrow \mathcal{D}(NewImage(P,S_i), NewImage(Q,S_i)) \geq \\ &\quad \mathcal{D}(Image(P,S_i), Image(Q,S_i)) \qquad (8) \end{aligned}$$

Furthermore, the difference between the two (i.e. $\mathcal{D}(NewImage(P,S_i), NewImage(Q,S_i))$ and $\mathcal{D}(Image(P,S_i), Image(Q,S_i)))$ is usually significant when computing the distance of the query from a point in the cluster: Say, P is a point in S_i and Q is the query point. Due to the reconstruction distance bound, $ReconDist(P,S_i,\mathcal{D})$ is *always* a small number ($\leq MaxReconDist$). On the other hand, $ReconDist(Q,S_i,\mathcal{D})$ can have any arbitrary value and is usually much larger than $ReconDist(P,S_i,\mathcal{D})$), thus making the difference quite significant. This makes the distance computations in the (d_i+1)-d more optimistic than that in the d_i-d index and hence a better estimate of the distances in the original D-d space. For example, for a range query, the range condition $(\mathcal{D}(NewImage(P,S_i), NewImage(Q,S_i)) \leq \rho)$ is more optimistic (i.e. satisfies fewer objects) than the range condition $(\mathcal{D}(Image(P,S_i), Image(Q,S_i)) \leq \rho)$, leading to fewer false positives. The same is true for k-NN queries. Fewer false positives imply lower query cost. At the same time, adding a new dimension also increases the cost of the query. Our experiments show that decrease in the query cost from fewer false positives offsets the increase of the cost of the adding a dimension, reducing the overall cost of the query significantly (cf. Section 5, Figure 12).

4.2 Query Processing over the Global Index

In this section, we discuss how to execute similarity queries efficiently using the index structure described above (cf. Figure 5). We describe the query processing algorithm for point, range and k-NN queries. For correctness, the query processing algorithm must guarantee that it always returns exactly the same answer as the query on the original space [7]. Often dimensionality reduction techniques do not satisfy the correctness criteria [14, 19].

We show that all our query processing algorithms satisfy the above criteria.

4.2.1 Point Search

To find an object O, we first find the cluster that contains O. It is the first cluster S (in the order mentioned in Step FC6) for which the reconstruction distance bound is satisfied. If such a cluster S exists, we compute $NewImage(O, S)$ and find it in the corresponding index by invoking the point search algorithm of the index structure. The point search returns the object if it exists in the cluster, otherwise it returns null. If no such cluster S exists, O must be, if at all, in $\mathcal{O}$. So we sequentially search through $\mathcal{O}$ and return it if it exists in $\mathcal{O}$.

4.2.2 Range Queries

A range query $\mathcal{Q} = \langle Q, \rho, \mathcal{D}\rangle$ retrieves all objects O in the database that satisfies the range condition $\mathcal{D}(Q, O) \leq \rho$. The algorithm proceeds as follows (see [6] for pseudocode). For each cluster S_i, we map the query anchor Q to its (d_i+1)-d image Q_i (using the principal components Φ_i and subspace dimensionality d_i stored in the root node R of GI) and execute a range query (with the same range ρ) on the corresponding cluster index I_i by invoking the procedure RangeSearchOnClusterIndex on the root node R_i of I_i. RangeSearchOnClusterIndex is the standard R-tree-style recursive range search procedure that starts from the root node and explores the tree in a depth-first fashion. It examines the current node T: if T is a non-leaf node, it recursively searches each child node N of T that satisfies the condition $MINDIST(Q, N, \mathcal{D}) \leq \rho$ (where $MINDIST(Q, N, \mathcal{D})$ denotes the minimum distance of the $(d_i + 1)$-d image of query point to the $(d_i + 1)$-d bounding rectangle of N based on distance function $\mathcal{D}$); if T is a leaf node, it retrieves each data item O stored in T (which is the $NewImage$ of the original D-d object) [5] that satisfies the range condition $\mathcal{D}(Q, O) \leq \rho$ in the $(d_i + 1)$-d space, accesses the full D-dimensional tuple on disk to determine whether it is a false positive and adds it to the result set if it is not a false positive (i.e. it also satisfies the range condition $\mathcal{D}(Q, O) \leq \rho$ in the original D-d space). After all the cluster indices are searched, we add all the qualifying points from among the outliers to the result by performing a sequential scan on $\mathcal{O}$. Since the distance in the index space lower bounds the distance in the original space (cf. Lemma 1), the above algorithm cannot have any false dismissals. The algorithm cannot have any false positives either as they are filtered out before adding to the result set. The above algorithm thus returns exactly the same answer as the query on the original space.

4.2.3 k Nearest Neighbor Queries

A k-NN query $\mathcal{Q} = \langle Q, k, \mathcal{D}\rangle$ retrieves a set $\mathcal{R}$ of k objects such that for any two objects $O \in \mathcal{R}, O' \notin \mathcal{R}, \mathcal{D}(Q, O) \leq \mathcal{D}(Q, O')$. The algorithm for k-NN queries is shown in Table 4. Like the basic k-NN algorithm, the algorithm uses a priority queue $queue$ to navigate the nodes/objects in the database in increasing order of their distances from Q. Note that we use a single queue to navigate the entire global index i.e. we explore the nodes/objects of all the cluster indices in an intermixed fashion and do not require separate queues to navigate the different clusters. Each entry in $queue$ is either a node or an object and stores 3 fields: the id of the node/object T it corresponds to, the cluster S it belongs to and its distance $dist$ from the query anchor Q. The items (i.e. nodes/objects) are prioritized based on $dist$ i.e. the smallest item appears at the top of the queue (min-priority queue). For nodes, the distance is defined by $MINDIST$ while for objects, it is the the point-to-point distance. Initially, for each cluster, we map the query anchor Q to its (d_i+1)-d image Q_i using the information stored in the root node R of GI (Line 2). Then, for each cluster index I_i, we compute the distance $MINDIST(Q_i, R_i, \mathcal{D})$ of Q_i from the root node R_i of I_i and push R_i into $queue$ along with the distance and the id of the cluster S_i to which it belongs (Line 3). We also fill the set $temp$ with the k closest neighbors of Q among the outliers by sequentially scanning through $\mathcal{O}$ (Line 4).

```
k-NNSearch(Query $\mathcal{Q} = Q, k, \mathcal{D}\rangle$)

1   for (i=1; i $\leq K$; i++)
2       $Q_{S_i} \leftarrow$ NewImage(Q, $S_i$);
3       $queue$.push($S_i, R_i, MINDIST(Q_i, R_i, \mathcal{D})$);
4   Add to $temp$ the $k$ closest neighbors of $Q$ among $\mathcal{O}$ (lin. scan)
5   while (not $queue$.IsEmpty())
6       top=queue.Top();
7       for each object O in $temp$ such that $O.dist \leq top.dist$
8           $temp \leftarrow temp - O$;
9           $result$ = $result \cup$ O;
10          retrieved++;
11          if (retrieved = k) return $result$;
12      queue.Pop();
13      if $top.T$ is an object
14        $top.dist = \mathcal{D}(Q, Original(top.T, top.S))$;
15        $temp$ = $temp \cup top.T$;
16      else if $top.T$ is a leaf node
17        for each object $O$ in $top.T$
18            $queue$.push(top.S, O, $\mathcal{D}(Q_{top.S}, O)$);
19      else /* $top.T$ is an index node */
20        for each child $N$ of $top.T$
21            $queue$.push(top.S, $N$, $MINDIST(Q_{top.S}, N, \mathcal{D})$);
```

Table 4: k-NN Query.

After these initialization steps, we start navigating the index by popping the item from the top of $queue$ at each step (Line 11). If the popped item is an object, we compute the distance of the original D-d object (by accessing the full tuple on disk) from Q and append it to $temp$ (Lines 12-14). If it a node, we compute the distance of each of its children to the appropriate query image $Q_{top.S}$ (where $top.S$ denotes the cluster which top belongs to) and push them into the queue (Lines 15-20). Note that the image for each cluster is computed just once (in Step 2) and is reused here. We move an object O from $temp$ to $result$ only when we are sure that it is among the k nearest neighbors of Q i.e. there exists no object $O' \notin result$ such that $\mathcal{D}(O', Q) < \mathcal{D}(O, Q)$ and $|result| < k$. The second condition is ensured by the exit condition in Line 11. The condition $O.dist \leq top.dist$ in Line 7 ensures that there exists no *unexplored* object O' such that $\mathcal{D}(O', Q) < \mathcal{D}(O, Q)$. The proof is simple: $O.dist \leq top.dist$ implies $O.dist \leq \mathcal{D}(NewImage(O', S), NewImage(Q, S))$

[5]Note that instead of storing the 'NewImage's, we could have stored the original D-d points in the leaf pages of the cluster indices (in both cases, the index is built on the reduced space). Our choice of the former option is explained in [6].

for any unexplored object O' in a cluster S (by the property of min-priority queue) which in turn implies $\mathcal{D}(O,Q) \leq \mathcal{D}(O',Q)$ (since $D(NewImage(O',S), NewImage(Q,S))$ lower bounds $\mathcal{D}(O',Q)$, see Lemma 1). By inserting the objects in $temp$ (i.e. already explored items) into $result$ in increasing order of their distances in the original D-d space (by keeping $temp$ sorted), we also ensure there exists no *explored* object O' such that $\mathcal{D}(O',Q) < \mathcal{D}(O,Q)$. This shows that the algorithm returns the correct answer i.e. the exact set of objects as the query in the original D-d space. It is also easy to show that the algorithm is I/O optimal.

Lemma 2 (Optimality of k-NN algorithm) *The k-NN algorithm is optimal i.e. it does not explore any object outside the range of kth nearest neighbor. (Proof in [6]).*

4.3 Modifications

We assume that the data is static in order to build the index. However, we must support subsequent insertions/deletions of the objects to/from the index efficiently. We do not describe the insertion and deletion algorithms in this paper due to space limitations but they can be found in [6].

5 Experiments

In this section, we present the results of an extensive empirical study we have conducted to (1) evaluate the effectiveness of LDR as a high dimensional indexing technique and (2) compare it with other techniques, namely, GDR, original space indexing (OSI) and linear scan. We conducted our experiments on both synthetic and real-life datasets. The major findings of our study can be summarized as follows:

- **High Precision:** LDR provides up to an order of magnitude improvement in precision over the GDR technique at the same reduced dimensionality. This indicates that LDR can achieve the same reduction as GDR with significantly lower loss of distance information.
- **Low Query Cost:** LDR consistently outperforms other indexing techniques, namely GDR, original space indexing and sequential scan, in terms of query cost (combined I/O and CPU costs) for both synthetic and real-life datasets.

Thus, our experimental results validate the thesis of this paper that LDR is an effective indexing technique for high dimensional datasets. All experiments reported in this section were conducted on a Sun Ultra Enterprise 450 machine with 1 GB of physical memory and several GB of secondary storage, running Solaris 2.5.

5.1 Experimental Methodology

We conduct the following two sets of experiments to evaluate the LDR technique and compare it with other indexing techniques.

Precision Experiments

Due to dimensionality reduction, both GDR and LDR, cause loss of distance information. More the number of dimensions eliminated, more the amount of information lost. We measure this loss by *precision* defined as $Precision = \frac{|R_{original}|}{|R_{reduced}|}$ where $R_{reduced}$ and $R_{original}$ are the sets of answers returned by the range query on the reduced dimensional space and the original HDFS respectively [14]. We repeat that since our algorithms guarantee that the user always gets back the correct set $R_{original}$ of answers (as if the query executed in the original HDFS), precision does *not* measure the quality of the answers returned to the user but just the information loss incurred by the DR technique and hence the query cost. For a DR technique, if we fix the reduced dimensionality, the higher the precision, the lower the cost of the query, the more efficient the technique. We compare the GDR and LDR techniques based on precision at fixed reduced dimensionalities.

Cost Experiments

We conducted experiments to measure the query cost (I/O and CPU costs) for each of the following four indexing techniques. We describe how we compute the I/O and CPU costs of the techniques below.

- *Linear Scan:* In this technique, we perform a simple linear scan on the original high dimensional dataset. The I/O cost in terms of sequential disk accesses is $\frac{N*(D*sizeof(float)+sizeof(id))}{PageSize}$. Since $sizeof(id) \ll (D * sizeof(float))$, we will ignore the $sizeof(id)$ henceforth. Assuming sequential I/O is 10 times faster than random I/O, the cost in terms of the random accesses is $\frac{N*sizeof(float)*D)}{10*PageSize}$. The CPU cost is the cost of computing the distance of the query from each point in the database.
- *Original Space Indexing (OSI):* In this technique, we build the index on the original HDFS itself using a multidimensional index structure. We use the hybrid tree as the index structure. The I/O cost (in terms of random disk accesses) of the query is the number of nodes of the index structure accessed. The CPU cost is the CPU time (excluding I/O wait) required to navigate the index and return the answers.
- *GDR:* In this technique, we peform PCA on the original dataset, retain the first few principal components (depending on the desired reduced dimensionality) and index the reduced dimensional space using the hybrid tree index structure. In this case, the I/O cost has 2 components: index page accesses (discussed in OSI) and accessing the full tuples in the relation for false positive elimination (post processing cost). The post processing cost can be one I/O per false positives in the worst case. However, as observed in [11], this assumption is overly pessimistic (and is confirmed by our experiments). We, therefore, assume the postprocessing I/O cost to be $\frac{num\_false\_positives}{2}$. The total I/O cost (in number of random disk accesses) is $index\_page\_access\_cost + \frac{num\_false\_positives}{2}$. The CPU cost is the sum of the index CPU cost and the post processing CPU cost i.e. cost of computing the distance of the query from each of the false positives.
- *LDR:* In this technique, we index each cluster using the hybrid tree multidimensional index structure and used a linear scan for the outlier set. For LDR, the I/O cost of a query has 3 components: index page accesses for each cluster index, linear scan on the outlier set and accessing the full tuples in the relation (post processing cost). The total index page access cost is the total number of nodes accessed of all the cluster indices combined. The number of sequential disk accesses for the outlier scan is $\frac{|\mathcal{O}|*D*sizeof(float)}{PageSize}$. The cost of outlier scan in terms of random accesses is $\frac{|\mathcal{O}|*sizeof(float)*D)}{10*PageSize}$.

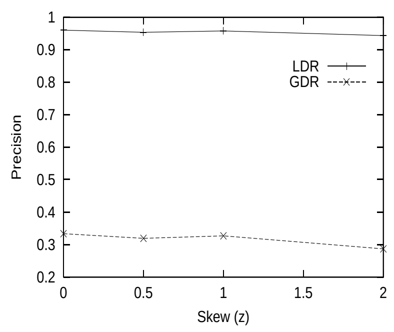

Figure 6: Sensitivity of precision to skew.

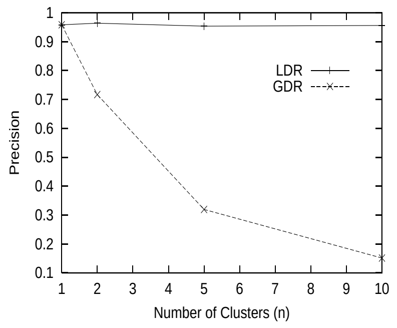

Figure 7: Sensitivity of precision to number of clusters.

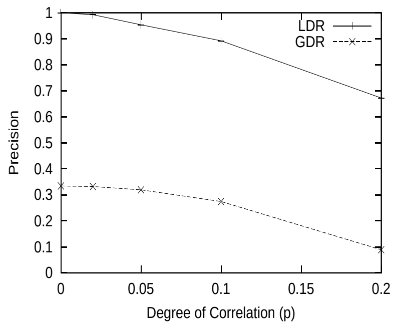

Figure 8: Sensitivity of precision to degree of correlation.

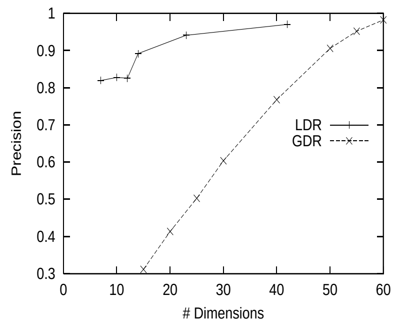

Figure 9: Sensitivity of precision to reduced dimensionality.

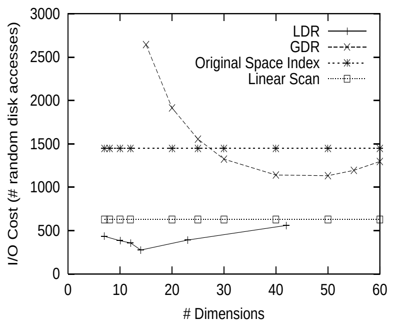

Figure 10: Comparison of LDR, GDR, Original Space Indexing and Linear Scan in terms of I/O cost. For linear scan, the cost is computed as: $\frac{num\_sequential\_disk\_accesses}{10}$.

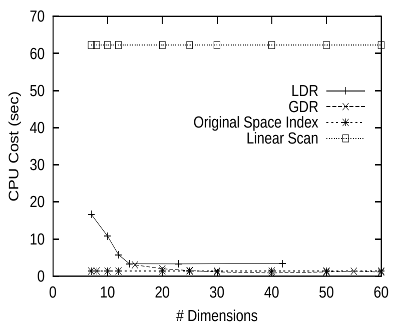

Figure 11: Comparison of LDR, GDR, Original Space Indexing and Linear Scan in terms of CPU cost.

The postprocessing I/O cost is $\frac{num\_false\_positives}{2}$ (as discussed above). The total I/O cost (in number of random disk accesses) is $index\_page\_access\_cost + \frac{|\mathcal{O}|*sizeof(float)*D)}{10*PageSize} + \frac{num\_false\_positives}{2}$. Similarly, the CPU cost is the sum of the index CPU cost, outlier scan CPU cost (i.e. cost of computing the distance of the query from each of the outliers) and the post processing cost (i.e. cost of computing the distance of the query from each of the false positives).

We chose the hybrid tree as the index structure for our experiments since it is a space partitioning index structure ("dimensionality-independent" fanout) and has been shown to scale to high dimensionalities [5]. [6] We use a page size of 4KB for all our experiments.

5.2 Experimental Results - Synthetic Data Sets

Synthetic Data Sets and Queries

In order to generate the synthetic data, we use a method similar to that discussed in [21] but appropriately modified so that we can generate the different clusters in subspaces of different orientations and dimensionalities. The synthetic dataset generator is described in Appendix A. The dataset generated has original dimensionality of 64 and consists of 100,000 points. The input parameters to the data generator and their default values are shown in Table 5 (Appendix A).

We generated 100 range queries by selecting their query anchors randomly from the dataset and choosing a range value such that the average query selectivity is about 2%. We tested with only range queries since the k-NN algorithm, being optimal, is identical to the range query with the range equal to the distance of the kth nearest neighbor from the query (Lemma 3). We use L_2 distance (Euclidean) as the distance metric. All our measurements are averaged over the 100 queries.

Precision Experiments

In our first set of experiments, we carry out a sensitivity analysis of the GDR and LDR techniques to parameters like skew in the size of the clusters (z_{size}), number of clusters (k) and degree of correlation (p). In each experiment, we vary the parameter of interest while the remaining parameters are fixed at their default values. We fix the reduced dimensionality of the GDR technique to 15. We fix the average subspace dimensionality of the clusters (i.e. $\Sigma_{i=1}^{K} \frac{n_i d_i}{K}$) also to 15 by choosing $FracOutliers$ and $MaxReconDist$ appropriately ($FracOutliers = 0.1$ and $MaxReconDist = 0.5$). Figure 6 compares the precision of the LDR technique with that of GDR for various value of z_{size}. LDR achieves about 3 times higher precision compared to GDR i.e. the latter has more than three times the number of false positives as the former. The precision of neither technique changes significantly with the skew. Figure 7 compares the precision of the two techniques for various values of k. As expected, for one cluster, the two techniques are identical. As k increases, the precision of GDR deteriorates while that of LDR is indepen-

[6]The performance gap between our technique and the other techniques was even greater with SR-tree [15] as the index structure due to higher dimensionality curse [5]. We do not report those results here but can be found in the full version of the paper [6].

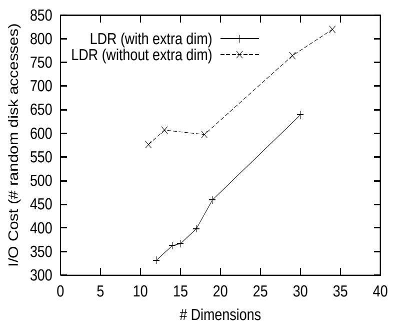

Figure 12: Effect of adding the extra dimension.

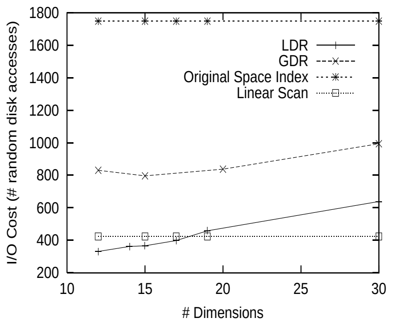

Figure 13: Comparison of LDR, GDR, Original Space Indexing and Linear Scan in terms of I/O cost. For linear scan, the cost is computed as: $\frac{num\_sequential\_disk\_accesses}{10}$.

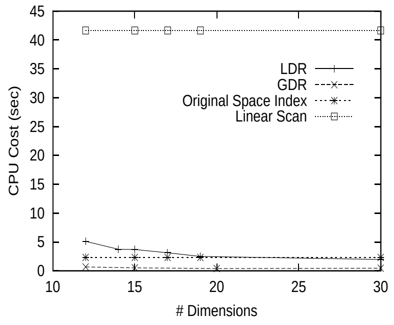

Figure 14: Comparison of LDR, GDR, Original Space Indexing and Linear Scan in terms of CPU cost.

dent of the number of clusters. For $k = 10$, LDR is almost an order of magnitude better compared to GDR in terms of precision. Figure 8 compares the two techniques for various values of p. As the degree of correlation decreases (i.e. the value of p increases), the precision of both techniques drop but LDR outperforms GDR for all values p. Figure 9 shows the variation of the precision with the reduced dimensionality. For the GDR technique, we vary the reduced dimensionality from 15 to 60. For the LDR technique, we vary the $FracOutliers$ from 0.2 to 0.01 (0.2, 0.15, 0.1, 0.05, 0.02, 0.01) causing the average subspace dimensionality to vary from 7 to 42 (7, 10, 12, 14, 23 and 42) ($MaxDim$ was 64). The precision of both techniques increase with the increase in reduced dimensionality. Once again, LDR consistently outperforms GDR at all dimensionalities. The above experiments show that LDR is a more effective dimensionality reduction technique as it can achieve the same reduction as GDR with significantly lower loss of information (i.e. high precision) and hence significantly lower cost as confirmed in the cost experiments described next.

Cost Experiments

We compare the 4 techniques, namely LDR, GDR, OSI and Linear Scan, in terms of query cost for the synthetic dataset. Figure 10 compares the I/O cost of the 4 techniques. Both the LDR and GDR techniques have U-shaped cost curves: when the reduced dimensionality is too low, there is a high degree of information loss leading to a large number of false positives and hence a high post-processing cost; when it is too high, the index page access cost becomes too high due to dimensionality curse. The optimum points lies somewhere in the middle: it is at dimensionality 14 (about 250 random disk accesses) for LDR and at 40 (about 1200 random disk accesses) for GDR. The I/O cost of OSI and Linear Scan is obviously independent of the reduced dimensionality. LDR significantly outperforms all the other 3 techniques in terms of I/O cost. The only technique that comes close to LDR in terms of I/O cost is the linear scan (but LDR is 2.5 times better as the latter performs 6274 sequential accesses $\sim$ 627 random accesses). However, linear scan loses out mainly due to its high CPU cost shown in Figure 11. While LDR, GDR and OSI techniques have similar CPU cost (at their respective optimum points), the CPU cost linear scan is almost two orders of magnitude higher that the rest. LDR has slightly higher CPU cost compared to GDR and OSI since it uses linear scan for the outlier set: however, the savings in the I/O cost over GDR and OSI (by a factor of 5-6) far offsets the slightly higher CPU cost.

5.3 Experimental Results - Real-Life Data Sets

Description of Dataset

Our real-life data set (COLHIST dataset [5]) comprises of 8×8 color histograms (64-d data) extracted from about 70,000 color images obtained from the Corel Database (http://corel.digitalriver.com/) and is available online at the UCI KDD Archive web site (http://kdd.ics.uci.edu/databases/CorelFeatures). We generated 100 range queries by selecting their query anchors randomly from the dataset and choosing a range value such that the average query selectivity is about 0.5%. All our measurements are averaged over the 100 queries.

Cost Experiments

First, we evaluate the impact of adding $ReconDist$ as an additional dimension of each cluster in the LDR technique. Figure 12 shows that the additional dimension reduces the cost of the query significantly. We performed the above experiment on the synthetic dataset as well and observed a similar result. [7] Figure 13 compares the 4 techniques, namely LDR, GDR, OSI and Linear Scan, in terms of I/O cost. LDR outperforms all other techniques significantly. Again, the only technique that come close to LDR in I/O cost (i.e. number of random disk accesses) is the linear scan. However, again, linear scan turns out to significantly worse compared to LDR in terms of the overall cost due to its high CPU cost as shown in Figure 14.

6 Conclusion

With numerous emerging applications requiring efficient access to high dimensional datasets, there is a need for scalable techniques to indexing high dimensional data. In this paper, we proposed local dimensionality reduction (LDR) as an approach to indexing high dimensional spaces. We developed an algorithm to discover the locally correlated clusters in the dataset and perform dimensionality reduction on each of them individually. We presented an index structure that exploits the correlated clusters to efficiently support similarity queries over high dimensional datasets. We have shown that our query processing algorithms

[7]We also analyzed the sensitivity of the LDR technique to the $MaxReconDist$ parameter. The results can be found in [6].

are correct and optimal. We conducted an extensive experimental study with synthetic as well as real-life datasets to evaluate the effectiveness of our technique and compare it to GDR, original space indexing and linear scan techniques. Our results demonstrate that our technique (1) reduces the dimensionality of the data with significantly lower loss in distance information compared to GDR, outperforming GDR by almost an order of magnitude in terms of query precision (for the same reduced dimensionality) and (2) significantly outperforms all the other 3 techniques (namely, GDR, original space indexing and linear scan) in terms of the query cost for both synthetic and real-life datasets.

7 Acknowledgements

We thank David Eppstein and Padhraic Smyth for the useful discussions on the clustering algorithm. We thank Kriengkrai Porkaew for the discussions and his help with the implementation. We thank Corel Corporation for making the large collection of images used in the COLHIST dataset available to us. Our PCA implementation is built on top of the Meschach Library downloaded from `http://www.netlib.org/c/meschach/`.

References

[1] R. Agarwal, J. Gehrke, D. Gunopolos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. *Proc. of SIGMOD*, 1998.

[2] C. Aggarwal, C. Procopiuc, J. Wolf, P. Yu, and J. Park. Fast algorithms for projected clustering. *Proc. of SIGMOD*, 1999.

[3] S. Berchtold, D. A. Keim, and H. P. Kriegel. The x-tree: An index structure for high-dimensional data. *Proc. of VLDB*, 1996.

[4] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is “nearest neighbor” meaningful? *Proc. of ICDT*, 1998.

[5] K. Chakrabarti and S. Mehrotra. The hybrid tree: An index structure for high dimensional feature spaces. *Proceedings of the IEEE International Conference on Data Engineering*, March 1999.

[6] K. Chakrabarti and S. Mehrotra. Local dimensionality reduction: A new approach to indexing high dimensional spaces. *Technical Report, TR-MARS-00-04, University of California at Irvine, http://www-db.ics.uci.edu/pages/publications/*, 2000.

[7] C. Faloutsos, W. Equitz, M. Flickner, W. Niblack, D. Petkovic, and R. Barber. Efficient and effective querying by image content. In *Journal of Intelligent Information Systems, Vol. 3, No. 3/4*, pages 231–262, July 1994.

[8] C. Faloutsos and K.-I. D. Lin. Fastmap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proc. ACM SIGMOD*, pages 163–174, May 1995.

[9] K. Fukunaga. *Introduction to Statistical Pattern Recognition*. Academic Press, second edition edition, 1990.

[10] T. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 1985.

[11] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, 1993.

[12] S. Guha, R. Rastogi, and K. Shim. Cure: An efficient clustering algorithm for large databases. *Proc. of SIGMOD*, 1998.

[13] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Conf., pp. 47–57.*, 1984.

[14] K. V. R. Kanth, D. Agrawal, and A. K. Singh. Dimensionality reduction for similarity searching dynamic databases. *Proc. of SIGMOD*, 1998.

[15] N. Katayama and S. Satoh. The sr-tree: An index structure for high dimensional nearest neighbor queries. *Proc. of SIGMOD*, 1997.

[16] K. Lin, H. V. Jagadish, and C. Faloutsos. The TV-tree - an index stucture for high dimensional data. In *VLDB Journal*, 1994.

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

[18] R. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. *Proc. of VLDB*, 1994.

[19] M. Thomas, C. Carson, and J. Hellerstein. Creating a customized access method for blobworld. *Proc. of ICDE*, 2000.

[20] R. Weber, H. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high dimensional spaces. *Proc. of VLDB*, 1998.

[21] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: An efficient data clustering method for very large databases. *Proc. of SIGMOD*, 1996.

A Synthetic Data Generation

Param.	Description	Default Value
n	Total number of points	100000
D	Original dimensionality	64
k	Number of clusters	5
d	Avg. subspace dimensionality	10
z_{dim}	Skew in subspace dim. across clusters	0.5
z_{size}	Skew in size across clusters	0.5
c	Number of spatial cluster per cluster	10
r	Extent (from centroid) along subspace dim	0.5
p	Max displacement along non-subspace dim	0.1
o	Fraction outliers	0.05

Table 5: Input parameters to Synthetic Data Generator

The generator generates k clusters with a total of $n.(1-o)$ points distributed among them using a Zipfian distribution with value z_{size}. The subspace dimensionality of each cluster also follows a Zipfian distribution with value z_{dim}, the average subspace dimensionality being d. Each cluster is generated as follows. For a cluster with size n_i and subspace dimensionality d_i (computed using the Zipfian distributions described above), we randomly choose d_i dimensions among the D dimensions as the subspace dimensions and generate n_i points in that d_i-d plane. Along each of the remaining $(D-d_i)$ non-subspace dimensions, we assign a randomly chosen coordinate to all the n_i points in the cluster. Let f_j be the randomly chosen coordinate along the jth non-subspace dimension. In the subspace, the points are spatially clustered into several regions (c regions on average) with each region having a randomly chosen centroid and an extent of r from the centroid along each of the d_i dimensions. After all the points in the cluster are generated, each point is displaced by a distance of at most p in either direction along each non-subspace dimension i.e. the point is randomly placed somewhere between $(f_j - p)$ and $(f_j + p)$ along the jth non-subspace dimension. The amount of displacement (i.e. value of p) determines the degree of correlation (since r is fixed). Lower the value, more the correlation. To make the subspaces arbitrarily oriented, we generate a random orthonormal rotation matrix (generated using MATLAB) and rotate the cluster by multiplying the data matrix with the rotation matrix. After all the clusters are generated, we randomly generate $N.o$ points (with random values along all D dimensions) as the outliers. The default values of the various parameters is shown in Table 5.