Clustering for Approximate Similarity Search in High-Dimensional Spaces

Chen Li
Department of Computer Science
Stanford University
Stanford, CA 94306
chenli@db.stanford.edu

Edward Chang
Electrical & Computer Engineering
University of California
Santa Barbara, CA 93106
echang@ece.ucsb.edu

Hector Garcia-Molina
Department of Computer Science
Stanford University
Stanford, CA 94306
hector@db.stanford.edu

Gio Wiederhold
Department of Computer Science
Stanford University
Stanford, CA 94306
gio@db.stanford.edu

Abstract

In this paper we present a clustering and indexing paradigm (called Clindex) for high-dimensional search spaces. The scheme is designed for approximate similarity searches, where one wishes to find many of the data points near a target point, but where one can tolerate missing a few near points. For such searches, our scheme can find near points with high recall in very few IOs and perform significantly better than other approaches. Our scheme is based on finding clusters, and then

building a simple but efficient index for them. We analyze the trade-offs involved in clustering and building such an index structure, and present extensive experimental results.

Keywords: approximate search, clustering, high-dimensional index, similarity search.

1 Introduction

Similarity search has generated a great deal of interest lately because of applications such as similar text/image search and document/image copy detection. These applications characterize objects (e.g., images and text documents) as *feature vectors* in very high-dimensional spaces [13, 23]. A user submits a query object to a search engine, and the search engine returns objects that are similar to the query object. The degree of similarity between two objects is measured by some distance function between their feature vectors. The search is performed by returning the objects that are *nearest* to the query object in high-dimensional spaces.

Nearest-neighbor search is inherently expensive, especially when there are a large number of dimensions. First, the search space can grow exponentially with the number of dimensions. Second, there is simply no way to build an index on disk such that *all* nearest neighbors to *any* query point are physically adjacent on disk. (We discuss this "curse of dimensionality" in more detail in Section 2.) Fortunately, in many cases it is sufficient to perform an *approximate search* that returns many but not all nearest neighbors [2, 17, 15, 27, 29, 30]. (A feature vector is often an approximate characterization of an object, so we are already dealing with approximations anyway.) For instance, in content-based image retrieval [11, 19, 40] and document copy detection [9, 13, 20], it is usually acceptable to miss a small fraction of the target objects. Thus it is not necessary to pay the high price of an exact search.

In this paper we present a new similarity-search paradigm: a clustering/indexing combined scheme that achieves approximate similarity search with high efficiency. We call this approach *Clindex* (CLustering for INDEXing). Under Clindex, the dataset is first partitioned into "similar" clusters. To improve IO efficiency, each cluster is then stored in a sequential file, and a mapping table is built for indexing the clusters. To answer a query, clusters that are near the query point are retrieved into main memory. Clindex then ranks the objects in the retrieved clusters by their distances to the query

object, and returns the top, say k, objects as the result.

Both clustering and indexing have been intensively researched (we survey related work in Section 2), but these two subjects have been studied separately with different optimization objectives: clustering optimizes classification accuracy, while indexing maximizes IO efficiency for information retrieval. Because of these different goals, indexing schemes often do not preserve the clusters of datasets, and randomly project objects that are close (hence similar) in high-dimensional spaces onto a 2D plane (the disk geometry). This is analogous to breaking a vase (cluster) apart to fit it into the minimum number of small packing boxes (disk blocks). Although the space required to store the vase may be reduced, finding the boxes in a high-dimensional warehouse to restore the vase requires a great deal of effort.

In this study we show that by (1) taking advantage of the clustering structures of a dataset, and (2) taking advantage of sequential disk IOs by storing each cluster in a sequential file, we can achieve efficient approximate similarity search in high-dimensional spaces with high accuracy. We examine a variety of clustering algorithms on two different data sets to show that Clindex works well when (1) a dataset can be grouped into clusters and (2) an algorithm can successfully find these clusters. As a part of our study, we also explore a very natural algorithm called *Cluster Forming* (CF) that achieves a pre-processing cost that is linear in the dimensionality and polynomial in the dataset size, and can achieve a query cost that is independent of the dimensionality.

The contributions of this paper are as follows:

- We study how clustering and indexing can be effectively combined. We show how a rather natural clustering scheme (using grids as in [1]) can lead to a simple index that performs very well for approximate similarity searches.
- We experimentally evaluate the Clindex approach on a well-clustered $30,000$-image database. Our results show that Clindex achieves very high *recall* when the data can be well divided into clusters. It can typically return more than 90% of what we call the "golden" results (i.e., the best results produced by a linear scan over the entire dataset) with a few IOs.
- If the dataset does not have natural clusters, clustering may not be as effective. Fortunately, real datasets rarely occupy a large-dimensional space uniformly [4, 15, 38, 46, 48]. Our experiment with a set of $450,000$ randomly crawled Web images that do not have well-defined categories shows

that Clindex's effectiveness does depend on the quality of the clusters. Nevertheless, if one is willing to trade some accuracy for efficiency, Clindex is still an attractive approach.

- We also evaluate Clindex with different clustering algorithms (e.g., TSVQ), and compare Clindex with other traditional approaches (e.g., tree structures) to understand the gains achievable by pre-processing data to find clusters.
- Finally, we compare Clindex with PAC-NN [15], a latest approach for conducting approximate nearest neighbor search.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 presents Clindex and shows how a similarity query is conducted using our scheme. Section 4 presents the results of our experiments and compares the efficiency and accuracy of our approach with those of some traditional index structures. Finally, we offer our conclusions and discuss the limitations of Clindex in Section 5.

2 Related Work

In this section we discuss related work in three categories:

1: Tree-like index structures for similarity search,

2: Approximate similarity search, and

3: Indexing by clustering.

2.1 Tree-like Index Structures

Many tree structures have been proposed to index high-dimensional data (e.g., R*-tree [3, 24], SS-tree [45], SR-tree [28], TV-tree [31], X-tree [6], M-tree [16], and K-D-B-tree [36]). A tree structure divides the high-dimensional space into a number of subregions, each containing a subset of objects that can be stored in a small number of disk blocks. Given a vector that represents an object, a similarity query takes the following three steps in most systems [21]:

1. It performs a *where-am-I* search to find in which subregion the given vector resides.

2. It then performs a *nearest-neighbor* search to locate the neighboring regions where similar vectors may reside. This search is often implemented using a *range* search, which locates all the regions that overlap with the

search sphere, i.e., the sphere centered at the given vector with a diameter d.

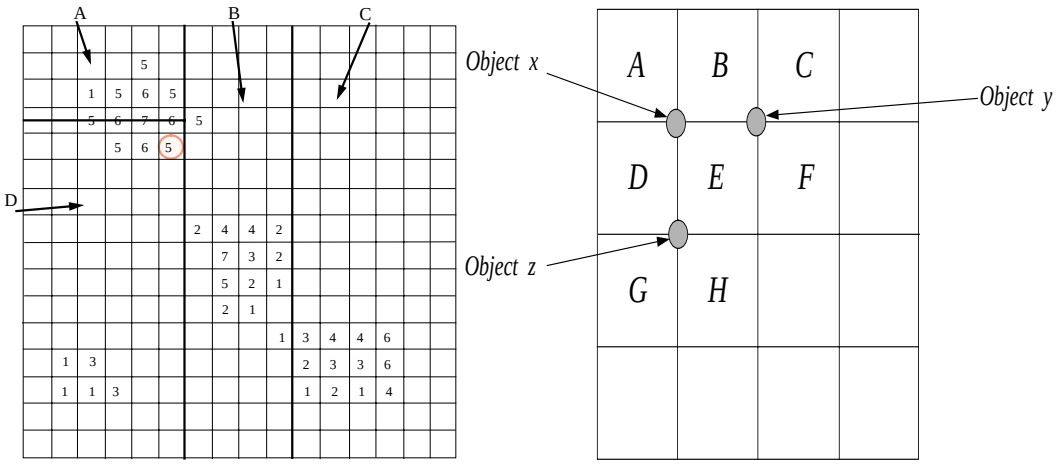
3. Finally, it computes the distances (e.g., Euclidean, street-block, or L^∞ distances) between the vectors in the nearby regions (obtained from the previous step) and the given vector. The search result includes all the vectors that are within distance d from the given vector.

The performance bottleneck of similarity queries lies in the first two steps. In the first step, if the index structure does not fit in main memory and the search algorithm is inefficient, a large portion of the index structure must be fetched from the disk. In the second step, the number of neighboring subregions can grow exponentially with respect to the dimension of the feature vectors. If D is the number of dimensions, the number of neighboring subregions can be on the order of $O(3^D)$ [21]. Roussopoulos et. al [37] propose the *branch-and-bound* algorithm and Hjaltason and Samet [25] propose the *priority queue* scheme to reduce the search space. But, when D is very large, the reduced number of neighboring pages to access can still be quite large. Berchtold et. al [5] propose the pyramid technique that partitions a high dimensional space into $2D$ pyramids and then cuts each pyramid into slices that are parallel to the basis of the pyramid. This scheme does not perform satisfactorily when the data distribution is skewed or when the search hypercube touches the boundary of the data space.

In addition to being copious, the IOs can be random and hence exceedingly expensive.[1] An example can illustrate what we call the *random-placement* syndrome faced by traditional index structures. Figure 1(a) shows a 2-dimensional Cartesian space divided into 16 equal stripes in both the vertical and the horizontal dimensions, forming a 16×16 grid structure. The integer in a cell indicates how many points (objects) are in the cell. Most index structures divide the space into subregions of equal points in a top-down manner. Suppose each disk block holds 40 objects. One way to divide the space is to first divide it into three vertical compartments (i.e., left, middle, and right), and then to divide the left compartment horizontally. We are left with four subregions A, B, C and D containing about the same number of points. Given a query object residing near the border of A, B and D, the similarity query has to retrieve blocks A, B and D. The number of subregions to check for the neighboring points grows exponentially with

[1] To transfer 100 KBytes of data on a modern disk with 8-KByte block size, doing a sequential IO is more than ten times faster than doing a random IO. The performance gap widens as the size of data transferred increases.

respect to the data dimension.



(a) Clustering by indexing

(b) Random placement

Figure 1: The shortcomings of tree structures.

Furthermore, since in high-dimensional spaces the neighboring subregions cannot be arranged in a manner sequential to all possible query objects, the IOs must be random. Figure 1(b) shows a 2-dimensional example of this random phenomenon. Each grid in the figure, such as A and B, represents a subregion corresponding to a disk block. The figure shows three possible query objects x, y and z. Suppose that the neighboring blocks of each query object are its four surrounding blocks. For instance, blocks A, B, D and E are the four neighboring blocks of object x. If the neighboring blocks of objects x and y are contiguous on disk, then the order must be $CFEBAD$, or $FCBEDA$, or their reverse orders. Then it is impossible to store the neighboring blocks of query object z contiguously on disk, and this query must suffer from random IOs. This example suggests that in high-dimensional spaces, the neighboring blocks of a given object must be dispersed randomly on disk by tree structures.

Many theoretical papers (e.g., [2, 27, 29]) have studied the cost of an exact search, independent of the data structure used. In particular, these papers show that if N is the size of a dataset, D is the dimension, and

$D >> log\ N$, then no nearest-neighbor algorithm can be significantly faster than a linear search.

2.2 Approximate Similarity Search

Many studies propose conducting approximate similarity search for applications where trading a small percentage of recall for faster search speed is acceptable. For example, instead of searching in all the neighboring blocks of the query object, study [44] proposes performing only the where-am-I step of a similarity query, and returning only the objects in the disk block where the query object resides. However, this approach may miss some objects similar to the query object. Take Figure 1(a) as an example. Suppose the query object is in the circled cell in the figure, which is near the border of regions A, B and D. If we return only the objects in region D where the query object resides, we miss many nearest neighbors in A.

Arya and Mount [2] suggest doing only ε-approximate nearest-neighbor searches, for $\varepsilon > 0$. Let d denote the function computing the distance between two points. We say that $p \in P$ is an ε-approximate nearest neighbor of q if for all $p' \in P$ we have $d(p,q) \leq (1+\varepsilon)d(p',q)$. Many follow-up studies have attempted to devise better algorithms to reduce search time and storage requirements. For example, Indyk and Motwani [27] and Kushilevitz et al. [30] give algorithms with polynomial storage and query time polynomial in $logn$ and d. Indyk and Motwani [27] also give another algorithm with smaller storage requirements and sublinear query time. Most of this work, however, is theoretical. The only practical scheme that has been implemented is the *locality-sensitive hashing* scheme proposed by Indyk and Motwani [27]. The key idea is to use hash functions such that the probability of collision is much higher for objects that are close to each other than for those that are far apart.

Approximate search has also been applied to tree-like structures. Recent work [15] shows that if one can tolerate $\varepsilon > 0$ relative error with a δ confidence factor, one can improve the performance of M-tree by 1-2 orders of magnitude.

Although an ε-approximate nearest-neighbor search can reduce the search space significantly, its recall can be low. This is because the candidate space for sampling the nearest neighbor becomes exponentially larger than the optimal search space. To remedy this problem, a follow-up study of [27] builds multiple *locality-preserving* indexes on the same dataset [26]. This is analogous to building n tree indexes on the same dataset, and each index

distributes the data into data blocks differently. To answer a query, one retrieves one block following each of the indexes and combines the results. This approach achieves better recall than is achieved by having only one index. But in addition to the n times pre-processing overhead, it has to replicate the data $n-1$ times to ensure that sequential IOs are possible via every index.

2.3 Clustering and Indexing by Clustering

Clustering techniques have been studied in the statistics, machine learning and database communities. The work in different communities focuses on different aspects of clustering. For instance, recent work in the database community includes CLARANS [35], BIRCH [48], DBSCAN [18], CLIQUE [1], and WaveClusters [39]. These techniques have high degree of success in identifying clusters in a very large dataset, but they do not deal with the efficiency of data search and data retrieval.

Of late, clustering techniques were explored for efficient indexing in high-dimensional spaces. Choubey, Chen and Rundensteiner [14] propose pre-processing data using clustering schemes before bulk-loading the clusters into an R-tree. They show that the bulk-loading scheme does not hurt retrieval efficiency compared to inserting tuples one at a time. Their bulk-loading scheme speeds up the inseration operation but is not designed to tackle the search efficiency problem that this study focuses on. For improving search efficiency, we presented the RIME system that we built [13] for detecting replicated images using an early version of Clindex [12]. In this paper, we extend our early work for handling approximate similarity search and conduct extensive experiments to compare our approach with others. Bennett, Fayyad and Geiger [4] propose using the EM (Expectation Maximization) algorithm to cluster data for efficient data retrieval in high-dimensional spaces. The quality of the EM clusters, however, depends heavily on the initial setting of the model parameters and the number of clusters. When the data distribution does not follow a Gaussian distribution and the number of clusters and the Gaussian parameters are not initialized properly, the quality of the resulting clusters suffers. How these initial parameters should be selected is still an open research problem [7, 8]. For example, it is difficult to know how many clusters exist before EM is run, but their EM algorithm needs this number to cluster effectively. Also, the EM algorithm may take many iterations to converge, and it may converge to one of many local optima [7, 32].

The clustering algorithm that Clindex uses in this paper does not make any assumption on the number of clusters nor on the distribution of the data. Instead, we use a bottom-up approach that groups objects adjacent in the space into a cluster. (This is similar to grouping stars into galaxias.) Therefore, a cluster can be in any shape. We believe that when the data is not uniformly distributed, this approach can preserve the natural shapes of the clusters. In addition, Clindex treats outliers differently by placing them in separate files. An outlier is not close to a cluster, so it is not similar to the objects in the clusters. However, an outlier can be close to other outliers. Placing outliers separately helps the search for the similar outliers more efficiently.

3 Clindex Algorithm

Since the traditional approaches suffer from a large number of random IOs, our design objectives are (1) to reduce the number of IOs, and (2) to make the IOs sequential as much as possible. To accomplish these objectives, we propose a clustering/indexing approach. We call this scheme Clindex (CLustering for INDEXing). The focus of this approach is to

- Cluster similar data on disk to minimize disk latency for retrieving similar objects, and
- Build an index on the clusters to minimize the cost of looking up a cluster.

To couple indexing with clustering, we propose using a common structure that divides space into grids. As we will show shortly, the same grids that we use for clustering are used for indexing. In the other direction, an insert/delete operation on the indexing structure can cause a regional reclustering on the grids. Clindex consists of the following steps:

1. It divides the i^{th} dimension into 2^κ stripes. In other words, at each dimension, it chooses $2^\kappa - 1$ points to divide the dimension. (We describe how to adaptively choose these dividing points in Section 3.3.2.) The stripes in turn define $(2^\kappa)^D$ cells, where D is the dimension of the search space. This way, using a small number of bits (κ bits in each dimension) we can encode to which cell a feature vector belongs.

2. It groups the cells into clusters. A cell is the smallest building block for constructing clusters of different shapes. This is similar to the idea in calculus of using small rectangles to approximate polynomial functions of any degree. The finer the stripes, the smaller the cells and the finer the building blocks that approximate the shapes of the clusters. Each cluster

is stored as a sequential file on disk.

3. It builds an index structure to refer to the clusters. A cell is the smallest addressing unit. A simple encoding scheme can map an object to a cell ID and retrieve the cluster it belongs to in one IO.

The remainder of this section is divided into four topics:

Clustering: Section 3.1 describes how Clindex works in its clustering phase.

Indexing: Section 3.2 depicts how a structure is built by Clindex to index clusters.

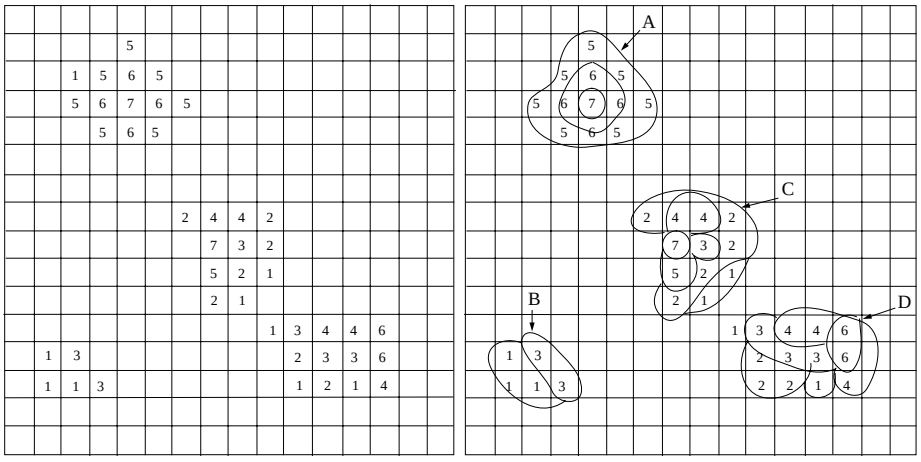
Tuning: Section 3.3 identifies all tunable control parameters and explains how changing these parameters affect Clidnex' performance.

Search: Finally, we show how a similar query is conducted using Clindex in Section 3.4.

3.1 The CF Algorithm: Clustering Cells

To perform efficient clustering in high-dimensional spaces, we use an algorithm called *cluster-forming* (CF). To illustrate the procedure, Figure 2(a) shows some points distributed on a 2D evenly-divided grid. The CF algorithm works in the following way:

1. CF first tallies the *height* (the number of objects) of each cell.
2. CF starts with the cells with the highest point concentration. These cells are the peaks of the initial clusters. (In the example in Figure 2(a), we start with the cells marked 7.)
3. CF descends one unit of the height after all cells at the current height are processed. At each height, a cell can be in one of three conditions: it is not adjacent to any cluster, it is adjacent to only one cluster, or it is adjacent to more than one cluster. The corresponding actions that CF takes are
   (a) If the cell is not adjacent to any cluster, the cell is the seed of a new cluster.
   (b) If the cell is adjacent to only one cluster, we join the cell to the cluster.
   (c) If the cell is adjacent to more than one cluster, the CF algorithm invokes an algorithm called *cliff-cutting* (CC) to determine



(a) Before clustering (b) After clustering

Figure 2: The grid before and after CF.

to which cluster the cell belongs, or if the clusters should be combined.

4. CF terminates when the height drops to a threshold, which is called the *horizon*. The cells that do not belong to any cluster (i.e., that are below the horizon) are grouped into an *outlier* cluster and stored in one sequential file.

Figure 2(b) shows the result of applying the CF algorithm to the data presented in Figure 2(a). In contrast to how the traditional indexing schemes split the data (shown in Figure 1(a)), the clusters in Figure 2(b) follow what we call the "natural" clusters of the dataset.

The formal description of the CF algorithm is presented in Figure 3. The input to CF includes the dimension (D), the number of bits needed to encode each dimension (κ), the threshold to terminate the clustering algorithm (θ), and the dataset (P). The output consists of a set of clusters (Φ) and a heap structure (H) sorted by cell ID for indexing the clusters. For each cell that is not empty, we allocate a structure C that records the cell

The Cluster Forming Algorithm

- **Input**:
  - D, κ, θ, P;
- **Output**:
  - Φ; /* cluster set */
  - H; /* heap */
- **Variables**:
  - τ, ψ, S;
- **Execution Steps**:

0: Init:
$\psi \leftarrow 0$; $\Phi \leftarrow \emptyset$; $S \leftarrow \emptyset$;

1: for each $p \in P$

1.1: $\tau \leftarrow Cell(p)$; /* Map point p to a cell id */

1.2: $C \leftarrow HeapFind(H, \tau)$

1.3(a): if $(C \neq nil)$
$C.\#p \leftarrow C.\#p + 1$;

1.3(b): else
new C;
$C.id \leftarrow \tau$; $C.\#p \leftarrow 1$;
$HeapInsert(H, C)$;

2: $S \leftarrow \{C | C \in H\}$; /* S is a temp array holding a copy of all cells */

3: $Sort(S)$; /* sort cells in descending order on $C.\#p$ */

4: $i \leftarrow 0$; $C \leftarrow S[i]$;

5: while $((C \neq nil)$ and $(C.\#p \geq \theta))$

5.1: $\Psi \leftarrow FindNeighborClusters(S, C.id)$ /* Ψ contains cell C's neighboring clusters */

5.2(a): if $(\Psi = \emptyset)$ /* not adjacent to any cluster */
new β; /* β holds a new cluster structure */
$\beta.C \leftarrow C.id$;
$\Phi \leftarrow \Phi \cup \{\beta\}$; /* Insert the new cluster into the cluster set */
$C.\beta \leftarrow \beta$; /* Update to which cluster the cell belongs */

5.2(b): else If $(|\Psi| \geq 1)$ /* If the cell is adjacent to one or more than one cluster */
$C.\beta \leftarrow CC(C, \Psi, \Phi, H, S)$; /* CC is described in Section 3.1.2 */

5.3: $i \leftarrow i + 1$; $C \leftarrow S[i]$;

6: new β; $\beta.C \leftarrow 0$; /* Group remaining cells into an outlier cluster */

7: $\Phi \leftarrow \Phi \cup \{\beta\}$;

8: for $(C = S[i]; C \neq nil; i++)$ $C.\beta \leftarrow \beta$;

Figure 3: The Cluster Forming (CF) algorithm.

ID ($C.id$), the number of points in the cell ($C.\#p$), and the cluster that the cell belongs to ($C.\beta$). The cells are inserted into the heap. CF is a two-pass algorithm. After the initialization step (step 0), its first pass (step 1) tallies the number of points in each cell. For each point p in the data set P, it maps the point into a cell ID by calling the function *Cell*. The function *Cell* divides each dimension into 2^κ regions. Each value in the feature vector is replaced by an integer between 0 and $2^\kappa - 1$, which depends on where the value falls in the range of the dimension. The quantized feature vector is the cell ID of the object. The CF algorithm then checks whether the cell exists in the heap (by calling the procedure $HeapFind$, which can be found in a standard algorithm book). If the cell exists, the algorithm increments the point count for the cell. Otherwise, it allocates a new cell structure, sets the point count to one, and inserts the new cell into the heap (by calling the procedure $HeapInsert$, also in standard textbooks).

In the second pass, the CF algorithm clusters the cells. In step 2 in the figure, CF copies the cells from the heap to a temporary array S. Then, in steps 3 and 4 it sorts the cells in the array in descending order on the point count ($C.\#p$). In step 5, the algorithm checks if a cell is adjacent to some existing clusters starting from the cell with the greatest height down to the termination threshold θ. If a cell is not adjacent to any existing cluster, a new cluster β is formed in step 5.2(a). The CF algorithm records the centroid cell for the new cluster in $\beta.C$ and inserts the cluster into the cluster set Φ. If the cell is adjacent to more than one cluster, the algorithm calls the procedure CC (cliff cutting) in step 5.2(b) to determine which cluster the cell should join. (Procedure CC is described in Section 3.1.2.) The cell then joins the identified (new or existing) cluster. Finally, in steps 6 to 8, the cells that are below the threshold are grouped into one cluster as outliers.

3.1.1 Time Complexity:

We now analyze the time complexity of the CF algorithm. Let N denote the number of objects in the dataset and M be the number of nonempty cells. First, it takes $O(D)$ to compute the cell ID of an object and it takes $O(D)$ time to check whether two cells are adjacent. During the first pass of the CF algorithm, we can use a heap to keep track of the cell IDs and their heights. Given a cell ID, it takes $O(D \times log\ M)$ time to locate the cell in the heap. Therefore, the time complexity of the first phase is $O(N \times D \times logM)$.

During the second pass, the time to find all the neighboring cells is $O(min\{3^D, M\})$. The reason is that there are at most three neighboring

stripes of a given cell in each dimension. We can either search all the neighboring cells, which are at most $3^D - 1$, or search all the nonempty cells, which are at most M. In a high-dimensional space[2], we believe that $M << 3^D$. Therefore, the time complexity of the second phase is $O(D \times M^2)$. The total time complexity is $O(N \times D \times log\ M) + O(D \times M^2)$.

Since the clustering phase is done off-line, the pre-processing time may be acceptable in light of the speed that is gained in query performance.

3.1.2 Procedure CC

In the Cliff-Cutting (CC) procedure, we need to decide to which cluster the cell belongs. Many heuristics exist. We choose the following two rules:

1: If the new object is adjacent to only one cluster, then add it to the cluster.

2: If the new object is adjacent to more than one cluster, we

- Merge all clusters adjacent to the cell into one cluster.
- Insert this cell to the cluster.

Note that the above rules may produce large clusters. One can avoid large clusters by increasing either κ or θ as we discuss in Sections 3.3.1 and 3.3.2.

3.2 The Indexing Structure

In the second step of the CF algorithm, an indexing structure is built to support fast access to the clusters generated. As shown in Figure 4, the indexing structure includes two parts: a *cluster directory* and a *mapping table*. The cluster directory keeps track of the information about all the clusters, and the mapping table maps a cell to the cluster where the cell resides.

All the objects in a cluster are stored in sequential blocks on disk, so that these objects can be retrieved by efficient sequential IOs. The cluster directory records the information about all the clusters, such as the cluster ID, the name of the file that stores the cluster, and a flag indicating whether the cluster is an outlier cluster. The cluster's centroid, which is the center of all the cells in the cluster, is also stored. With the information in the

[2]Take image databases as an example. Many image databases use feature vectors with more than 100 dimensions. Clearly, the number of images in an image database is much smaller than 3^{100}.

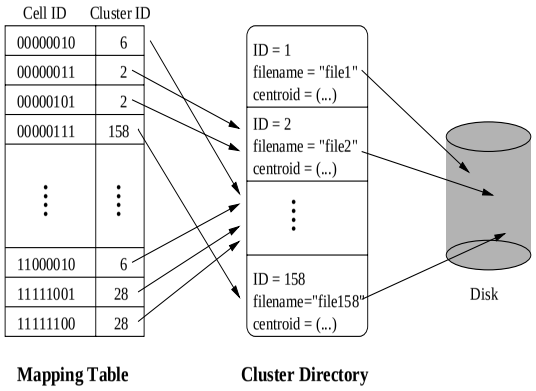


Figure 4: The index structure.

cluster directory, the objects in a cluster can be retrieved from disk once we know the ID of the cluster.

The mapping table is used to support fast lookup from a cell to the cluster where the cell resides. Each entry has two values: a cell ID and a cluster ID. The number of entries in the mapping table is the number of nonempty cells (M), and the empty cells do not take up space. In the worst case, we have one cell for each object, so there are at most N cell structures. The ID of each cell is a binary code with the size $D \times \kappa$ bits, where D is the dimension and κ is the number of bits we choose for each dimension. Suppose that each cluster ID is represented as a two-byte integer. The total storage requirement for the mapping table is $M \times (D \times \kappa/8 + 2)$ bytes. In the worst case, $M = N$, and the total storage requirement for the mapping table is on the order of $O(N \times D)$. The disk storage requirement of the mapping table is comparable to that of the interior nodes in a tree index structure.

Note that the cell IDs can be sorted and stored sequentially on disk. Given a cell ID, we can easily search its corresponding entry by doing a binary search. Therefore, the number of IOs to look up a cluster is $O(log\ M)$, which is comparable the cost of doing a where-am-I search in a tree index structure.

3.3 Control Parameters

The granularity of the clusters is controlled by four parameters:

- D: the number of data dimensions or object features.
- κ: the number of bits used to encode each dimension.
- N: the number of objects.
- θ: the horizon.

The number of cells is determined by parameters D and κ and can be written as $2^{D \times \kappa}$. The average number of objects in each cell is $\frac{N}{2^{D \times \kappa}}$. This means that we are dealing with two conflicting objectives. On the one hand, we do not want to have low point density, because low point density results in a large number of cells but a relatively small number of points in each cell and hence tends to create many small clusters. For similarity search you want to have a sufficient number of points to present a good choice to the requester, say, at least seven points in each cluster [33]. On the other hand, we do not want to have densely-populated cells either, since having high point density results in a small number of very large clusters, which cannot help us tell objects apart.

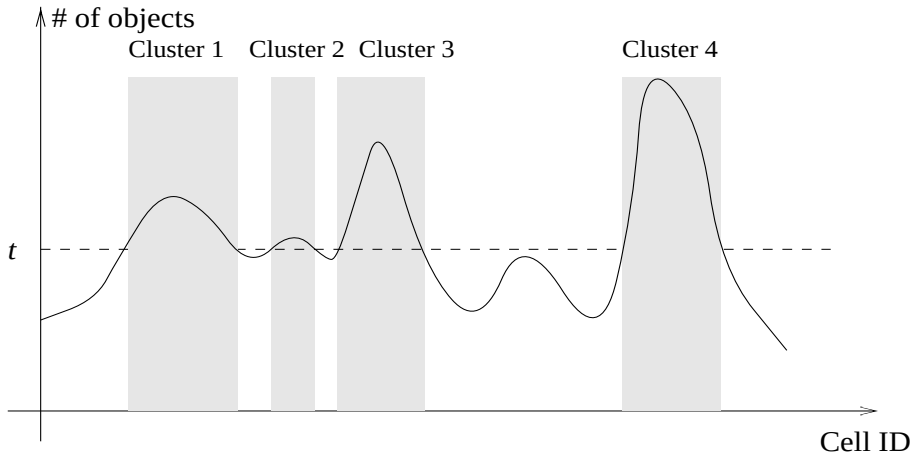


Figure 5: A Clustering Example.

3.3.1 Selecting θ

The value of θ affects the number and the size of the clusters. Figure 5 shows an example in a one-dimensional space. The horizontal axis is the cell IDs and the vertical axis the number of points in the cells. The θ value set at the t level threshold forms four clusters. The cells whose heights are below $\theta = t$ are clustered into the outlier cluster. If the threshold is reduced, both the cluster number and the cluster size may change, and the size of the outlier cluster decreases. If the outlier cluster has a relatively small number of objects, then it can fit into a few sequential disk blocks to improve its IO efficiency. On the other hand, it might be good for the outlier cluster to be relatively large, because then it can keep the other clusters well separated.

The selection of a proper θ value can be quite straightforward in an indirect way. First, one decides what percentage of data objects should be outliers. We can then add to the 5^{th} step of the CF algorithm (in Figure 3) a termination condition that places all remaining data into the outlier cluster once the number of remaining data objects drops below the threshold.

3.3.2 Selecting κ

Due to the uneven distribution of the objects, it is possible that some regions in the feature space are sparsely populated and others densely populated. To handle this situation, we need to be able to perform *adaptive clustering*.

Suppose we divide each dimension into 2^κ stripes. We can divide regions that have more points into smaller substripes. This way, we may avoid very large clusters, if this is desirable. In a way, this approach is similar to the extensible hashing scheme: for buckets that have too many points, the extensible hashing scheme splits the buckets. In our case, we can build clusters adaptively with different resolutions by choosing the dividing points carefully based on the data distribution. For example, for image feature vectors, since the luminescence is neither very high nor very low in most pictures, it makes sense to divide the luminescence spectrum coarsely at the two extremes and finely in the middle. This adaptive clustering step can be done in Step 1.1 in Figure 3. When the *Cell* procedure quantizes the value in each dimension, it can take the statistical distribution of the dataset in that dimension into consideration. This step, however, requires an additional data analysis pass before we execute the CF algorithm, so that the *Cell* procedure has the information to determine how to quantize each dimension properly.

To summarize, CF is a bottom-up clustering algorithm that can approximate the cluster boundaries to any fine detail and is adaptive to data distributions. Since each cluster is stored contiguously on disk, a cluster can be accessed with much more efficient IOs than traditional index-structure methods.

3.4 Similarity Search

Given a query object, a similarity search is performed in two steps. First, Clindex maps the query object's feature vector into a binary code as the ID of the cell where the object resides. It then looks up in the mapping table to find the entry of the cell. The search takes the form of different actions, depending on whether or not the cell is found:

- If the cell is found, we obtain the cluster ID to which the cell belongs. We find the file name where the cluster is stored in the cluster directory, and then read the cluster from disk into memory.
- If the cell is not found, the cell must be empty. We then find the cluster closest to the feature vector by computing and comparing the distances from the centroids of the clusters to the query object. We read the nearest cluster into main memory.

If a high recall is desirable, we can read in more nearby clusters. After we read candidate objects in these nearby clusters into main memory, we sort them according to their distances to the query object, and return the nearest objects to the user.

Remarks:

Our search can return more than one cluster whose centroid is close to the query object by checking the cluster directory. Since the number of clusters is much smaller than the number of objects in the dataset, the search for the nearest clusters can very likely be done via an in-memory lookup. If the number of clusters is very large, one may consider treating cluster centroids as points and apply clustering algorithms on the centroids. This way, one can build a hierarchy of clusters and narrow down the search space to those clusters (in a group of clusters) that are nearest to the query object. Another alternative is to precompute the k-nearest clusters and store their IDs in each cluster. As we show in Section 4, returning the top few clusters can achieve very high recall with very little time.

Example:

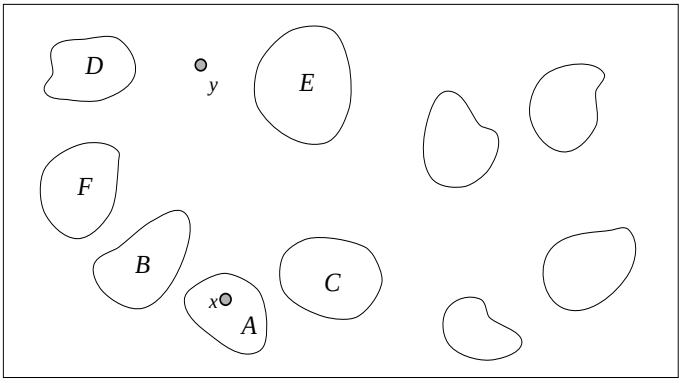


Figure 6: A Search Example.

Figure 6 shows a 2D example of how a similarity search is carried out. In the figure, A, B, C, D and E are five clusters. The areas not covered by these clusters are grouped into an outlier cluster. Suppose that a user submits x as the query object. Since the cell of object x belongs to cluster A, we return the objects in cluster A and sort them according to their distances to x. If a high recall is required, we return more clusters. In this case, since clusters B and C are nearby, we also retrieve the objects in these two clusters. All the objects in clusters A, B and C are ranked based on their distances to object x, and the nearest objects are returned to the user.

If the query object is y, which falls into the outlier cluster, we first retrieve the outlier cluster. By definition, the outlier cluster stores all outliers, and hence the number of points in the outlier that can be close to y is small. We also find the two closest clusters D and E, and return the nearest objects in these two clusters.

4 Evaluation

In our experiments we focused on queries of the form "find the top k most similar objects" or k Nearest Neighbors (abbreviated as k-NN). For each k-NN query, we return the top k nearest neighbors of the query object. To establish the benchmark to measure query performance, we scanned the entire dataset for each query object to find its top 20 nearest neighbors, the

"golden" results. There are at least three metrics of interest to measure the query result:

(a) Recall after X IOs: After X IOs are performed, what fraction of the k top golden results has been retrieved?

(b) Precision after X IOs: After X IOs, what fraction of the objects retrieved is among the top k golden results?

(c) Precision after $R\%$ of the top k golden objects has been found. We call this *R-precision*, and it quantifies how much "useful" work was done in obtaining some fraction of the golden results.

In this paper we focus on recall results, and comment only briefly on R-precision and relative distance errors [47]. In our environment, we believe that precision is not the most useful metric since the main overhead is IOs, not the number of non-golden objects that are seen.

We performed our experiments on two sets of images: a $30,000$ and a $450,000$ image set. The first set contains $30,000$ images from Corel CDs, which has been extensively used as a test-bed of content-based image retrieval systems by the computer vision and image processing communities. This $30,000$-image dataset consists images of different categories, such as landscapes, portraits, and buildings. The other dataset is a set of $450,000$ randomly-crawled images from the Internet.[3] This $450,000$-image dataset has a variety of content, ranging from sports, cartoons, clip arts, etc. and can be difficult to classify even manually. To characterize images, we converted each image to a 48-dimensional feature vector by applying the Daubechies' wavelet transformation [13].

In addition to using the CF clustering algorithm, we indexed these feature vectors using five other schemes: Equal Partition (EP), Tree-Structure Vector Quantization (TSVQ) [22], R*-tree, SR-tree, and M-tree with the PAC-NN scheme [15]:

- EP: To understand the role that clustering plays in Clindex, we devised a simple scheme, EP, that partitions the dataset into sequential files without performing any clustering. That is, we partitioned the dataset into cells with an equal number of images, where each cell occupies a contiguous region in the space and is stored in a sequential file. Since EP is very similar to Clindex with CF except for the clustering, any performance differences between the schemes must be due to the clustering. If the

[3] The readers are encouraged to experiment with the prototype, which is made available on-line [10]. We have been experimenting with our new approaches in image characterization so that the prototype may change from time to time.

differences are significant, it will mean that the dataset is not uniformly distributed and that Clindex with CF can exploit the clusters.

- TSVQ: To evaluate the effectiveness of Clindex's CF clustering algorithm, we replaced it with a more sophisticated algorithm, and then stored the clusters sequentially as usual. The replacement algorithm used is TSVQ, a k-mean algorithm [34], was implemented for clustering data in high-dimensional spaces. It has been widely used in data compression and lately in indexing high-dimensional data for content-based image retrieval [41, 42].
- R*-tree and SR-tree: Tree structures are often used for similarity searches in multidimensional spaces. To compare, we used the R*-tree and SR-tree structures implemented by Katayama and Satoh [28]. We studied the IO cost and recall of using these tree-like structures to perform similarity search approximately. Note that the comparison between Clindex and these tree-like structures will not be "fair" since neither R*-tree nor SR-tree performs off-line analysis (i.e., no clustering is done in advance). Thus, the results will only be useful to quantify the potential gains that the off-line analysis gives us, compared to traditional tree-based similarity search.
- M-tree with PAC-NN: We also compare Clindex with Ciaccia and Patella's PAC-NN implementation of the M-tree [15]. The PAC-NN scheme uses two parameters, ϵ and δ, to adjust the tradeoff between search speed and search accuracy. The *accuracy* ϵ allows for a certain relative error in the results, and the confidence δ guarantees, with probability at least $(1-\delta)$, that ϵ will not be exceeded.
  Since the PAC-NN implementation that we use supports only 1-NN search, we performed 20 1-NN queries to get 20 nearest neighbors. After each 1-NN query, the nearest neighbor found was removed from the the M-tree. After the 20^{th} 1-NN query, we had 20 nearest neighbors removed from the dataset and we compared them with the golden set to compute recall. Our test procedure is reliable for measuring recall for PAC-NN. However, our test procedure alters its search cost. It is well known that the cost of an exact 20-NN query is *not* 20 times the cost of a 1-NN query, but much less. For approximate queries, this seems also to be the case. To ensure fair comparison, we present the PAC-NN experimental results separately in Section 4.2.

As discussed in Section 3.4, Clindex always retrieves the cluster whose centroid is closest to the query object. We also added this intelligence to both TSVQ and EP to improve their recall. For R*-tree and SR-tree, how-

ever, we did not add this optimization.[4]

To measure recall, we used the cross-validation technique [34] commonly employed to evaluate clustering algorithms. We ran each test ten times, and each time we set aside 100 images as the test images and used the remaining images to build indexes. We then used these set-aside images to query the database built under four different index structures. We produced the average query recall for each run by averaging the recall of 100 image queries. We then took an average over 10 runs to obtain the final recall.

Our experiments are intended to answer the following questions:

1. How does clustering affect the recall of an approximate search? In Section 4.1 we perform different algorithms on two datasets that have different clustering quality to find out the effects of clustering.

2. How is the performance of Clindex compared to the PAC-NN scheme? In Section 4.2 we show how the PAC-NN scheme trades accuracy for speed in an M-tree structure.

3. How is the performance of Clindex in terms of elapsed time compared to the traditional structures and compared to sequentially reading in the entire file? In Section 4.3 we examine and compare how long the five schemes take to achieve a target recall.

4. How does block size affect recall? In Section 4.4 we cluster the $30,000$-image set using different block sizes to answer this question.

In the above experiments we collected the recall for 20-NN queries. In Section 4.5, we present the recalls of k-NN for $k = 1$ to 20.

4.1 Recall versus Clustering Algorithms

Figures 7 and 8 compare the recalls of CF, TSVQ, EP, R*-tree and SR-tree. We discuss the experimental results on the two data sets separately.

- The $30,000$-image dataset (Figure 7): In this experiment, all schemes divided the dataset into about 256 clusters. Given one IO to perform, CF returns the objects in the nearest cluster, which gives us an average of 62% recall (i.e., a return of 62% of the top 20 golden results). After we read three more clusters, the accumulated recall of CF increases to 90%.

[4] A scheme like R*-tree or SR-tree could be modified to pre-analyze the data and build a better structure. The results would be presumably similar to the results of EP, which keeps centroid information for each data block to assist effective nearest neighbor search. We did not develop the modified R*-tree or SR-tree scheme, and hence did not verify our hypothesis.

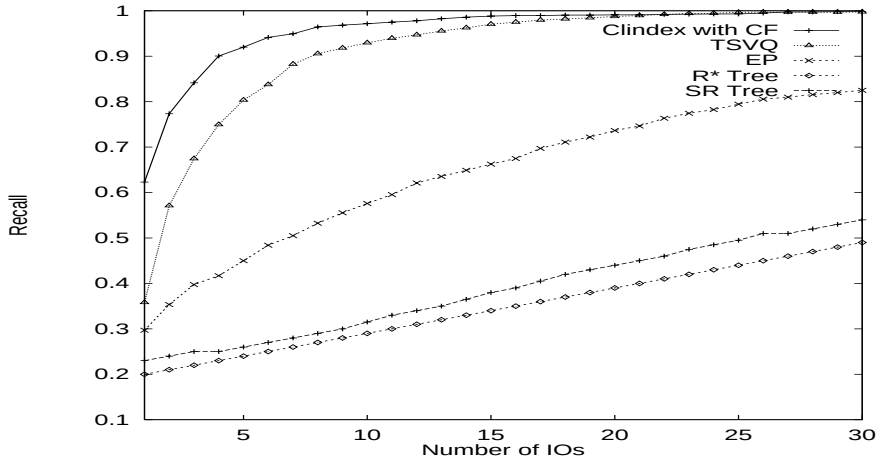


Figure 7: Recalls of Five Schemes (20-NN) on 30,000 Images.

If we read more clusters, the recall still increases but at a much slower pace. After we read 15 clusters, the recall approaches 100%. That is, we can selectively read in 6% ($\frac{15}{261} = 6\%$) of the data in the dataset to obtain almost all top 20 golden results.

The EP scheme obtains much lower recall than CF. It starts with 30% recall after the first IO is completed, and slowly progresses to 83% after 30 IOs. The TSVQ structure, although it does better than the EP scheme, still lags behind CF. It obtains 35% recall after one IO, and the recall does not reach 90% until after 10 IOs. The recall of CF and TSVQ converge after 20 IOs. Finally, R*-tree and SR-tree suffer from the lowest recall. (That SR-tree performs better than R*-tree is consistent with the results reported in [28].)

- The 450,000-image dataset (Figure 8): In this experiment, all schemes divided the dataset into about 1,500 clusters. Given one IO to perform, CF returns the objects in the nearest cluster, which gives us an average of 25% recall. After we read 10 more clusters, the accumulated recall of CF increases to 60%. After we read 90 clusters, the recall approaches 90%. Again, we can selectively read in 6% ($\frac{90}{1500} = 6\%$) of the data in the dataset to obtain 90% of the top 20 golden results. As expected, the achieved recall of this image dataset is not as good as that of the 30,000-image set. We believe that this is because the randomly crawled images may not form

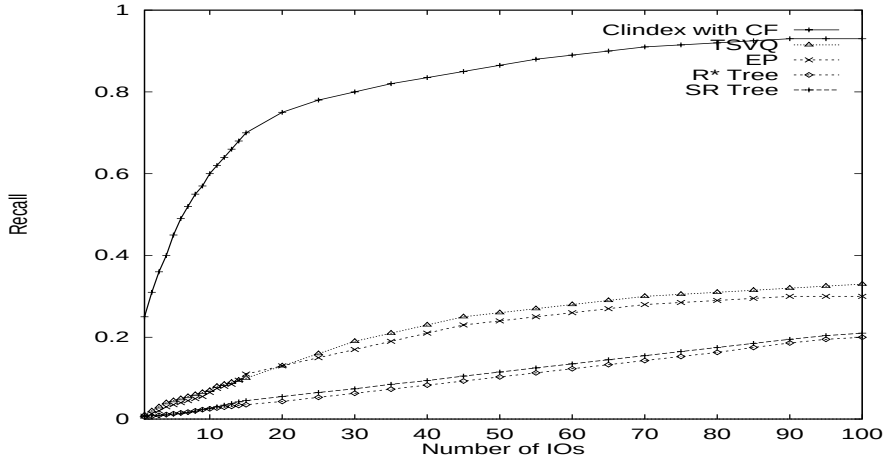


Figure 8: Recalls of Five Schemes (20-NN) on 450,000 Images.

clusters that are as well separated in the feature space as the clusters of the 30,000 image set. Nevertheless, in both cases the recall versus the percentage of data retrieved from the data set shows that CF can achieve a good recall by retrieving a small percentage of data.

The EP scheme achieves much lower recall than CF. It starts with 1% recall after the first IO is completed, and slowly progresses to 33% after 100 IOs. The TSVQ structure performs worse than the EP scheme. It achieves only 30% recall after 100 IOs. Finally, R*-tree and SR-tree suffer from the lowest recall. When the data does not display good clustering quality, a poor clustering algorithm exacerbates the problem.

Remarks

Using recall alone cannot entirely tell the quality of the approximate results. For instance, at a given recall, say 60%, the approximate results can be very different (in terms of distances) from the query point and can be fairly similar to the query. Figure 9 uses the *relative distance error* metric proposed in [47] to report the quality of the approximation. Let Q denote the query point in the feature space, D_G^k the average distance of the top-k golden results from Q in the feature space, and D_A^k the average distance of the top-k actual

results from Q. The relative distance error is measured by

$$\frac{D_A^k - D_G^k}{D_G^k}.$$

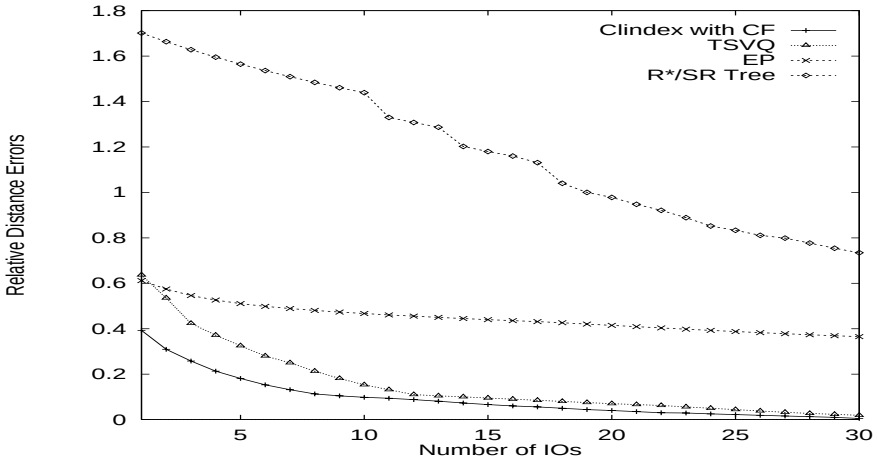


Figure 9: Relative Distance Errors (20-NN) on 30,000 Images.

Figure 9 shows that the relative distance errors of both Clindex and TSVQ can be kept under 20% after ten IOs. But the relative distance errors of the tree-structures are still quite large (more than 80%) even after 30 IOs.

4.2 The PAC-NN Scheme

We conducted PAC-NN experiments on the 30,000-image dataset. We observed that the value of δ plays a more important role than the value of ϵ in determining the search quality and cost. Table 1 shows one representative set of results where we set $\epsilon = 0.2$. (Setting ϵ to other values does not change the tradeoff pattern between search accuracy and cost.) Under five different δ settings, 0, 0.01, 0.02, 0.05 and 0.1, the table shows recall, number of IOs and wall-clock query time (for all 20 1-NN queries). (As we mentioned previously, the cost of 20 1-NN queries can be substantially higher than the cost of one 20-NN query. Therefore, one should only treat the number of IOs and the elapsed time in the table as loose bounds of the cost.)

When δ is increased from zero to 0.01, the recall of PAC-NN drops from 95.3% to 40%, but the number of IOs is reduced substantially to one eighth and the wall-clock elapsed time is reduced to one fifth. When we increase δ further, both the recall and cost continue to drop.

Remarks

- Quality of approximation: From the perspective of some other performance metrics that were proposed to measure quality of approximation [47], the PAC-NN scheme trades very slight accuracy for substantial search speedup. For multimedia applications where one may care more about getting similar results than finding the exact top-NN results, the PAC-NN scheme provides a scalable and effective way to trade accuracy for speed.
- IO cost: Since the PAC-NN scheme we experimented with is implemented with M-tree and a tree-like structure tends to distribute a nature cluster into many non-contiguous disk blocks (we have discussed this problem in Section 2), it takes the M-tree more IOs than Clindex to achieve a given recall. This observation is consistent with the results obtained from using other tree-like structures to index the images.
- Construction Time: It took only around 1.5 minutes to build an M-tree, while building Clindex with CF takes 30 minutes on a 600MHz Pentium-III workstation with 512 MBytes of DRAM. As we explained in Section 3, since the clustering phase is done off-line, the pre-processing time may be acceptable in light of the speed that is gained in query performance.

	$\delta = 0$	$\delta = 0.01$	$\delta = 0.02$	$\delta = 0.05$	$\delta = 0.1$
Recall	95.3%	40.0%	25.3%	12.2%	5.9%
Number of IOs	13,050	1,580	1,200	933	800
Wall-clock Time (seconds)	15	3	2	1.6	1.3

Table 1: The Performance and Cost of PAC-NN.

4.3 Running Time versus Recall

We have shown that clustering indeed helps improve recall when a dataset is not uniformly distributed in the space. This is shown by the recall gap between CF and EP in Figures 7 and 8. We have also shown that by

retrieving only a small fraction of data, CF can achieve a recall that is quite high (e.g., 90%). This section shows the time needed by the schemes to achieve a target recall. We compare their elapsed time to the time it takes to sequentially read in the entire dataset.

To report query time, we first use a quantitative model to compute IO time. Since IO time dominates a query time, this model helps explain why one scheme is faster than the others. We then compare the wall-clock time we obtained with the computed IO time. As we will show shortly, these two methods give us different elapsed times but the relative performance between the indexes are the same.

Let B denote the total amount of data retrieved to achieve a target recall, N the number of IOs, TR the transfer rate of the disk, and T_{seek} the average disk latency. The elapsed time T of a search can be estimated as

$$T = N \times T_{seek} + B/TR.$$

To compute T, we use the parameters of a modern disk listed in Table 2. To simplify the computation, we assume that an IO incurs one average seek (8.9 ms) and one half of a rotational delay (5.6 ms). We also assume that TR is 130 Mbps and that each image record, including its wavelets and a URL address, takes up 500 Bytes. For example, to compute the elapsed time to retrieve 4 clusters that contains 10% of the 30,000 images, we can express T as

$$T = 4 \times 14.5 + \frac{30,000 \times 10\% \times 500 \times 8}{130}\ ms.$$

Figure 10(a) shows the average elapsed time versus recall for five schemes on the 30,000-image dataset. Note that the average elapsed time is estimated based on the number of records retrieved in our ten rounds of experiments, not based on the number of clusters retrieved. The horizontal line in the middle of the figure shows the time it takes to read in the entire image feature file sequentially. To achieve 90% recall, CF and TSVQ take substantially less time than reading in the entire file. The traditional tree structures, on the other hand, perform worse than the sequential scan when we would like to achieve a recall that is above 65%.

Figure 10(b) shows the wall-clock elapsed time versus recall that we collected on a Dell workstation, which is configured with a 600MHz Pentium-III processor and 512 MBytes of DRAM. To eliminate the effect of caching, we ran ten queries on each index structure and we rebooted the system after

Parameter Name	*Value*
Disk Capacity	*26 GBytes*
Number of cylinders, CYL	*50,298*
Min. Transfer Rate TR	*130 Mbps*
Rotational Speed (RPM)	*5,400*
Full Rotational Latency Time	*11.12 ms*
Min. Seek Time	*2.0 ms*
Average Seek Time	*8.9 ms*
Max. Seek Time	*18 ms*

Table 2: Quantum Fireball Lct Disk Parameters

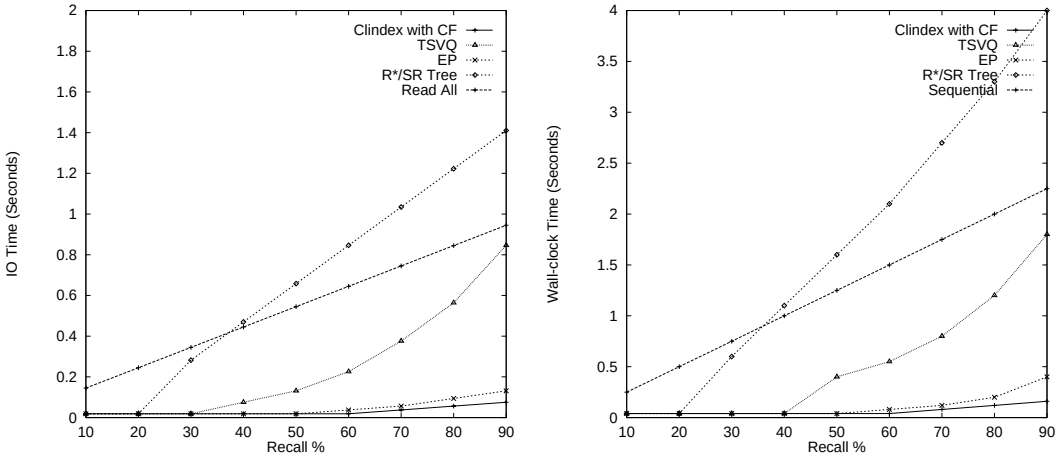
each query. The average running time of all schemes are about twice as long as the predicted IO time. We believe that this is because the wall-clock time includes index lookup time, distance computation time, IO bus time and memory access time, in addition to the IO time. The relative performance between indexes, however, are unchanged. In both figures, Clindex with CF requires the lowest running time to reach every recall target.

In short, we can draw the following conclusions:

1. Clindex with CF achieves a higher recall than TSVQ, EP and R*-tree, because CF is adaptive to wide variances of cluster shapes and sizes. TSVQ and other algorithms are forced to bound a cluster by linear functions, e.g., a cube in 3-D space. CF is not constrained in this way. Thus, if we have an odd cluster shape (e.g., with tentacles), CF will conform to the shape, while TSVQ will either have to pick a big space that encompasses all the "tentacles," or will have to select several clusters, each for a portion of the shape, and will thereby inadvertently break some "natural" clusters, as we illustrated in Figure 1(a). It is not surprising that the recall of TSVQ converges to that of CF after a number of IOs, because TSVQ eventually pieces together its "broken" clusters.

2. Using CF to approximate an exact similarity search requires reading just a fraction of the entire dataset. The performance is far better than that achieved by performing a sequential scan on the entire dataset.

3. CF, TSVQ and EP enjoy a boost in recall in their first few additional



(a) IO Time

(b) Wall-clock Time

Figure 10: Elapsed Time vs. Recall on 30,000 images.

IOs, since they always retrieve the next cluster whose centroid is the closest to the query object. R*-tree, conversely, does not store centroid information, and cannot selectively retrieve blocks to boost its recall in the first few IOs. (We have discussed the shortcomings of tree structures in Section 2 in detail.)

4. When a data set does not exhibit good clusters, CF's effectiveness can suffer. A poor clustering algorithm such as TSVQ and EP can suffer even more. Thus, employing a good clustering algorithm is important for Clindex to work well.

	Recall	60%	70%	80%	90%	$\approx 100\%$
	# of Golden Objects Retrieved	12	14	16	18	20
CF	*# of Objects Retrieved*	115	230	345	460	1,725
	R-Precision	10.44%	6.09%	4.64%	3.91%	1.16%
R-tree*	*# of Objects Retrieved*	2,670	3,430	4,090	4,750	5,310
	R-Precision	0.45%	0.41%	0.39%	0.379%	0.377%

Table 3: R-precision of 20-NN

Remarks on Precision

Figure 7 shows that while retrieving the same amount of data, CF has higher recall than R*-tree and other schemes. Put another way, CF also enjoys higher R-precision because retrieving the same number of objects from disk gives CF more golden results.

Since a tree structure is typically designed to use a small block size to achieve high precision, we tested R*-tree method with larger block size, and compared its R-precision with that of CF. We conducted this experiment only on the 30,000-image dataset. (The tree-like structures are ineffective on the 450,000-image dataset.) It took R*-tree 267 IOs on average to complete an exact similarity search. Table 3 shows that the R-precision of CF is more than ten times higher than that of R*-tree under different recall values.

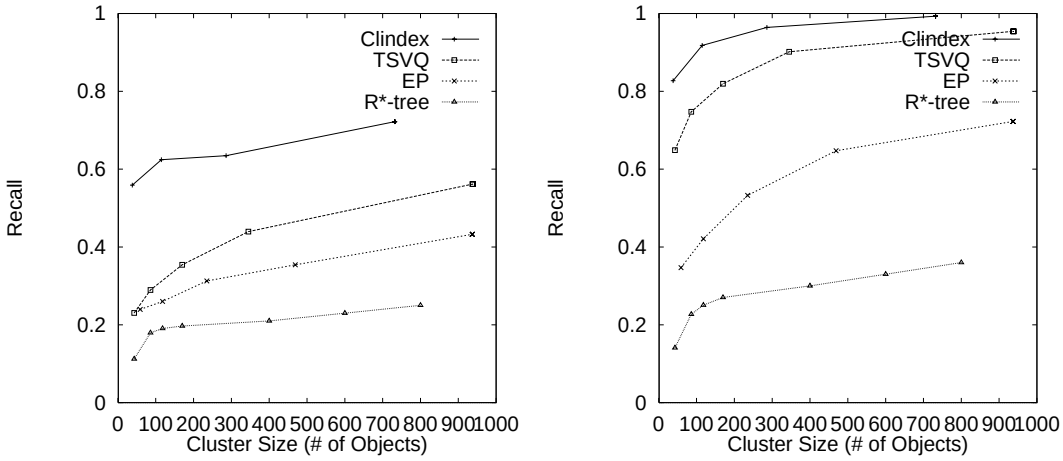
This result, although surprising, is consistent with that of many studies [28, 43]. The common finding is that when the data dimension is high, tree structures fail to divide points into neighborhoods and are forced to access almost all leaves during an exact similarity search. Therefore, the argument for using a small block size to improve precision becomes weaker as the data dimension increases.

4.4 Recall versus Cluster Size

Since TSVQ, EP and tree-like structures are ineffective on the 450,000-image dataset, we conducted the remaining experiments on the 30,000-image dataset only.

We define cluster size as the average number of objects in a cluster. To see the effects of the cluster size on recall, we collected recall values at different cluster sizes for CF, TSVQ, CF and R*-tree. Note that the cluster size of CF is determined by the selected values of κ and θ. We thus set four different κ and θ combinations to obtain recall values for four different cluster sizes. For EP, TSVQ and R*-tree, we selected cluster (block) sizes that contain from 50 up to 900 objects. Figure 11 depicts the recall (y-axis) of the four schemes for different cluster sizes (x-axis) after one IO and after five IOs are performed.

Figure 11(a) shows that given the same cluster size, CF has a higher recall than TSVQ, EP and R*-tree. The gaps between the clustering schemes (EP and R*-tree) and non-clustering schemes (TSVQ and CF) widen as the cluster size increases. We believe that the larger the cluster size, the more important a role the quality of the clustering algorithm plays. CF enjoys



(a) After one IO

(b) After five IOs

Figure 11: Recall versus Cluster Size.

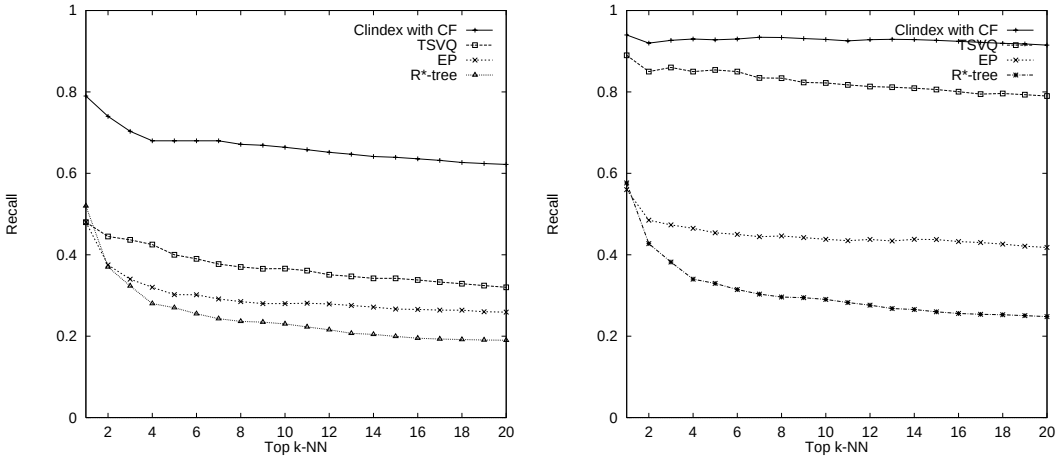
significantly higher recall than TSVQ, EP and R*-tree because it captures the clusters better than the other three schemes.

Figure 11(b) shows that CF still outperforms TSVQ, EP and R*-tree after five IOs. CF, TSVQ and EP enjoy better improvement in recall than R*-tree because, again, CF, TSVQ and EP always retrieve the next cluster whose centroid is nearest to the query object. On the other hand, a tree structure does not keep the centroid information, and the search in the neighborhood can be random and thus suboptimal. We believe that adding centroid information to the leaves of the tree structures can improve its recall.

4.5 Relative Recall versus k-NN

So far we measured only the recall of returning the top 20 nearest neighbors. In this section we present the recall when the top $k = 1$ to 19 nearest neighbors are returned after one and five IOs are performed. In these experiments, we partitioned the dataset into about 256 clusters (blocks) for all schemes, i.e., all schemes have about the same cluster size.

Figures 12(a) and Figures 12(b) present the relative recall relative to the top 20 golden results of the four schemes after one IO and five IOs are



(a) After One IO

(b) After Five IOs

Figure 12: Recall of k-NN.

performed, respectively. The x-axis in the figures represents the number of nearest neighbors requested, and the y-axis represents the relative recall. For instance, when only one nearest object is requested, CF returns the nearest object 79% of the time after one IO and 95% of the time after five IOs.

Both figures show that the recall gaps between schemes are insensitive to how many nearest neighbors are requested. We believe that once the nearest object can be found in the returned cluster(s), the conditional probability that additional nearest neighbors can also be found in the retrieved cluster(s) is high. Of course, for this conclusion to hold, the number of objects contained in a cluster must be larger than the number of nearest neighbors requested. In our experiments, a cluster contains 115 objects on average, and we tested up to 20 golden results. This leads us to the following discussion on how to tune CF's control parameters to form clusters of a proper size.

4.6 The Effects of the Control Parameters

One of Clindex's drawbacks is that its control parameters must be tuned to obtain quality clusters. We had to experiment with different values of

θ and κ to form clusters. We show here how we selected κ and θ for the $30,000$-image dataset.

Since the size of our dataset is much smaller than the number of cells ($2^{D \times \kappa}$), many cells are occupied by only one object. When we set $\theta \geq 1$, most points fell into the outlier cluster, so we set θ to zero. To test the κ values, we started with $\kappa = 2$. We increased κ by increments of one and checked the effect on the recall. Figure 13 shows that the recall with respect to the number of IOs decreases when κ is set beyond two. This is because in our dataset $D >> log\ N$, and using a κ that is too large spreads the objects apart and thereby leads to the formation of too many small clusters. By dividing the dataset too finely one loses the benefit of large block size for sequential IOs.

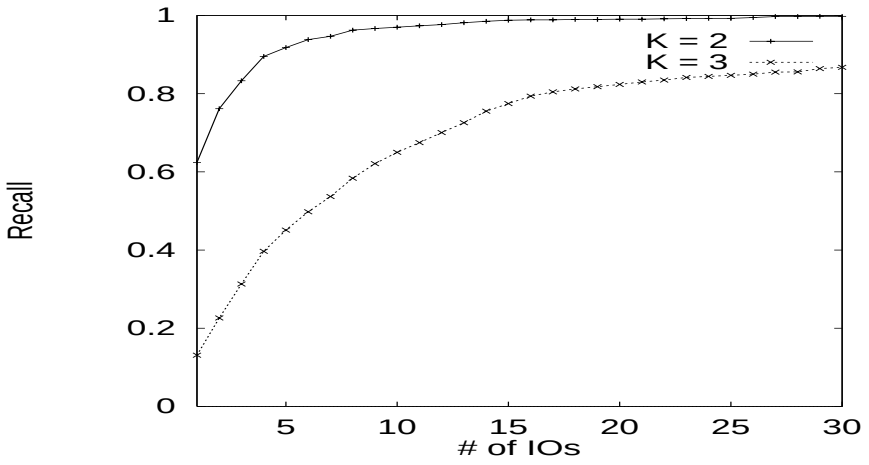


Figure 13: Recall versus κ.

Although the proper values of κ and θ are dataset dependent, we empirically found the following rules of thumb to be useful for finding good starting values:

- Choose a reasonable cluster size: The cluster must be large enough to take advantage of sequential IOs. It must also contain enough objects so that if a query object is near a cluster, then the probability that a significant number of nearest neighbors of the query object are in the cluster is high. On the other hand, a cluster should not be so large as to prolong query time unnecessarily. According to our experiments, increasing the cluster

size to where the cluster contains more than 300 objects (see Figure 11) does not further improve recall significantly. Therefore, a cluster of about 300 objects is a reasonable size.

- Determine the desired number of clusters: Once the cluster size is chosen, one can compute how many clusters the dataset will be divided into. If the number of clusters is too large to make the cluster table fit in main memory, we may either increase the cluster size to decrease the number of clusters, or consider building an additional layer of clusters.
- Adjust κ and θ: Once the desired cluster size is determined, we pick the starting values for κ and θ. The κ value must be at least two, so that the points are separated. The suggested θ is one, so that the clusters are separated. After running the clustering algorithm with the initial setting, if the average cluster is smaller than the desired size, we set θ to zero to combine small clusters. If the average cluster size is larger than the desired size, we can increase either κ or θ, until we obtain roughly the desired cluster size.

4.7 Summary

In summary, our experimental results show that:

1: Employing a better clustering algorithm improves recall as well as the elapsed time to achieve a target recall.

2: Providing additional information such as the centroids of the clusters helps prioritize the retrieval order of the clusters, and hence improves the search efficiency.

3: Using a large block size is good for both IO efficiency and recall.

Our experimental results also show that Clindex is effective when it uses a good clustering algorithm and when the data is not uniformly distributed. When a dataset is uniformly distributed in a high-dimensional space, sequentially reading in the entire dataset can be more effective than using Clindex.

5 Conclusions

In this paper, we presented Clindex, a paradigm for performing approximate similarity search in high-dimensional spaces to avoid the dimensionality curse. Clindex clusters similar objects on disk, and performs a similarity

search by finding the clusters near the query object. This approach improves the IO efficiency by clustering and retrieving relevant information sequentially on and from the disk. Its pre-processing cost is linear in D (dimensionality of the dataset), polynomial in N (size of the dataset) and quadratic in M (number of non-empty cells), and its query cost is independent of D. We believe that for many high-dimensional datasets (e.g., documents and images) that exhibit clustering patterns, Clindex is an attractive approach to support approximate similarity search both efficiently and effectively.

Experiments showed that Clindex typically can achieve 90% recall after retrieving a small fraction of data. Using the CF algorithm, Clindex's recall is much higher than that of some traditional index structures. Through experiments we also learned that Clindex works well because it uses large blocks, finds clusters more effectively, and searches the neighborhood more intelligently by using the centroid information. These design principles can also be employed by other schemes to improve their search performance. For example, one may replace the CF clustering algorithm in this paper with one that is more suitable for a particular dataset.

Finally, we summarize the limitations of Clindex and our future research plan.

- *Control parameter tuning*: As we discussed in Section 3.3, determining the value of θ is straightforward but determining a good κ value requires experiments on the dataset. We have provided some parameter-tuning guidelines in the paper. We plan to investigate a mathematical model which allows directly to determine the parameters. In this regard, we believe that the PAC-NN [15] approach can be very helpful.
- *Effective clustering*: Clindex needs a good clustering algorithm to be effective. There is a need to investigate the pros and cons of using existing clustering algorithms for indexing high-dimensional data.
- *Incremental clustering*: In addition, most clustering algorithms are off-line algorithms and the clusters can be sensitive to insertions and deletions. We plan to extend Clindex to perform regional reclustering for supporting insertions and deletions after initial clusters are formed.
- *Measuring performance using other metrics*: In this study, we use the classical precision and recall to measure the performance of similarity search. We plan to employ other metrics (e.g., [47]) to compare the performance between different indexing schemes.

Acknowledgments

We would like to thank James Ze Wang for his comments on our draft and his help on the TSVQ experiments. We would like to thank Kingshy Gho and Marco Patella for their assistance to complete the PAC-NN experiments on the M-tree structure. We would also like to thank the TKDE associate editor, Paolo Ciaccia, and reviewers for their helpful comments about the original version of this paper.

References

[1] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. *Proceedings of ACM SIGMOD*, June 1998.

[2] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Proceedings of the 5th SODA*, pages 573–82, 1994.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. *Proceedings of ACM SIGMOD*, May 1990.

[4] K. P. Bennett, U. Fayyad, and D. Geiger. Density-based indexing for approximate nearest-neighbor queries. *KDD*, 1999.

[5] S. Berchtold, C. Bohm, and H.-P. Kriegel. The pyramid-technique: Towards breaking the curse of dimensionality. *ACM Sigmod*, May 1998.

[6] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-Tree: An index structure for high-dimensional data. *Proceedings of the 22nd VLDB*, August 1996.

[7] C. Bishop. *Neural Networks for Pattern Recognition*. Oxford, 1998.

[8] P. S. Bradley, U. M. Fayyad, and C. A. Reina. Scaling em (expectation-maximization) clustering to large databases. *Microsoft Technical Report MSR-TR-98-34*, November 1999.

[9] S. Brin and H. Garcia-Molina. Copy detection mechanisms for digital documents. *Proceedings of ACM SIGMOD*, May 1995.

[10] E. Chang and T. Cheng. Pbir: Perception-based image retrieval. *UCSB Technical Report http:chang4.ece.ucsb.edu/∼echang/pbir/*, 2000.

[11] E. Chang, B. Li, and C. Li. Toward perception-based image retrieval (extended version). *UCSB Technical Report*, February 2000.

[12] E. Chang, C. Li, J. Wang, P. Mork, and G. Wiederhold. Searching near-replicas of images via clustering. *Proc. of SPIE Symposium of Voice, Video, and Data Communications*, September 1999.

[13] E. Chang, J. Wang, C. Li, and G. Wiederhold. RIME - a replicated image detector for the www. *Proc. of SPIE Symposium of Voice, Video, and Data Communications*, November 1998.

[14] R. Choubey, L. Chen, and E. A. Rundensteiner. Gbi: A geberalized r-tree bulk-insertion. *Proceedings of the* 8^{th} *SSD*, pages 91–108, July 1999.

[15] P. Ciaccia and M. Patella. Pac nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces. *Proceedings of ICDE*, pages 244–255, 2000.

[16] P. Ciaccia, M. Patella, and P. Zezula. M-Tree: An efficient access method for similarity search in metric spaces. *Proceedings of the 23rd VLDB*, August 1997.

[17] K. Clarkson. An algorithm for approximate closest-point queries. *Proceedings of the 10th SCG*, pages 160–64, 1994.

[18] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. *Proceedings of the 2nd International Conference on Knowledge Discovery in Databases and Data Mining*, August 1996.

[19] M. Flickner, H. Sawhney, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker. Query by image and video content: the QBIC system. *IEEE Computer*, 28(9):23–32, 1995.

[20] H. Garcia-Molina, S. Ketchpel, and N. Shivakumar. Safeguarding and charging for information on the internet. *Proceedings of ICDE*, 1998.

[21] H. Garcia-Molina, J. Ullman, and J. Widom. *Database System Implementation*. Prentice Hall, 1999.

[22] A. Gersho and R. Gray. *Vector Quantization and Signal Compression*. Kluwer Academic, 1991.

[23] A. Gupta and R. Jain. Visual information retrieval. *Comm. of the ACM*, 40(5):69–79, 1997.

[24] A. Guttman. R-trees: a dynamic index structure for spatial searching. *Proc. of ACM SIGMOD*, June 1984.

[25] G. R. Hjaltason and H. Samet. Ranking in spatial databases. *Proceedings of the* 4^{th} *SSD*, pages 83–95, August 1995.

[26] P. Indyk, A. Gionis, and R. Motwani. Similarity search in high dimensions via hashing. *Proceedings of the 25th VLDB*, September 1999.

[27] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. *Proceedings of the 30th STOC*, pages 604–13, 1998.

[28] N. Katayama and S. Satoh. The SR-Tree: An index structure for high-dimensional nearest neighbor queries. *Proceedings of ACM SIGMOD*, May 1997.

[29] J. M. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. *Proceedings of the 29th STOC*, 1997.

[30] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. *Proceedings of the 30th STOC*, pages 614–23, 1998.

[31] K.-L. Lin, H. V. Jagadish, and C. Faloutsos. The TV-tree: an index structure for high-dimensional data. *VLDB Journal*, 3(4), 1994.

[32] G. J. McLachlan and T. Krishnan. *The EM algorithm & extensions*. John Wiley & Sons, 1997.
[33] G. Miller. The magical number seven +- two, some limits on our capacity for processing information. *Psych Review*, 68:81–97, 1956.
[34] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.
[35] R. T. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. *Proceedings of the 20th VLDB*, September 1994.
[36] J. T. Robinson. The K-D-B-Tree: A search structure for large multidimensional dynamic indexes. *Proceedings of ACM SIGMOD*, April 1981.
[37] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. *ACM Sigmod*, pages 71–79, 1995.
[38] Y. Rubner, C. Tomasi, and L. Guibas. Adaptive color-image embedding for database navigation. *Proceedings of the the Asian Conference on Computer Vision*, January 1998.
[39] G. Sheikholeslami, S. Chatterjee, and A. Zhang. Wavecluster: A multi-resolution clustering approach for very large spatial databases. *Proceedings of the 34th VLDB Conference*, August 1998.
[40] J. R. Smith and S.-F. Chang. Visualseek: A fully automated content-based image query system. *ACM Multimedia Conference*, 1996.
[41] J. Z. Wang, G. Wiederhold, O. Firschein, and S. X. Wei. Wavelet-based image indexing techniques with partial sketch retrieval capability. *Proceedings of the 4th ADL*, May 1997.
[42] J. Z. Wang, G. Wiederhold, O. Firschein, and S. X. Wei. Content-based image indexing and searching using daubechies' wavelets. *Journal of Digital Libraries*, 1(4):311–28, 1998.
[43] R. Weber, H. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. *Proceedings of the 24th VLDB*, pages 194–205, 1998.
[44] D. A. White and R. Jain. Similarity indexing: Algorithms and performance. *Proc. SPIE Vol.2670, San Diego*, 1996.
[45] D. A. White and R. Jain. Similarity indexing with the SS-Tree. *Proceedings of the 12th ICDE*, Feb. 1996.
[46] G. Wiederhold. *Database Design (Second Edition)*. McGraw-Hill Book Company, 1983.
[47] P. Zezula, P. Savino, G. Amato, and F. Rabitti. Approximate similarity retrieval with m-trees. *VLDB Journal*, 7(4):275–293, December 1998.
[48] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: An efficient data clustering method for very large databases. *Proceedings of ACM SIGMOD*, June 1996.

Chen Li is a Ph.D. candidate in the Computer Science Department at Stanford University. His research interests include database systems and information technologies. In particular, he has been working on integration of heterogeneous information sources of structured, and semistructured data, clustering data in high-dimensional spaces and multimedia databases. From

Tsinghua University, Beijing, China, he received a BS in 1994 and an MS in 1996, both in Computer Science.

Edward Chang is an assistant professor in the Department of Electrical and Computer Engineering, and Computer Science at University of California, Santa Barbara. He received from Stanford University a MS in Computer Science in 1994, and a PhD in Electrical Engineering in 1999. From 1985 to 1995, he worked at Digital Equipment Corporation and Sun Microsystems, where he participated in product development and research in operating systems, transaction management and workflow systems. His current research focus is on multimedia databases.

Hector Garcia-Molina is the Leonard Bosack and Sandra Lerner Professor in the Departments of Computer Science and Electrical Engineering at Stanford University, Stanford, California. He is the chairman of the Computer Science Department since January 1, 2001. From August 1994 to December 1997 he was the Director of the Computer Systems Laboratory at Stanford. From 1979 to 1991 he was on the faculty of the Computer Science Department at Princeton University, Princeton, New Jersey. His research interests include distributed computing systems and database systems. He received a BS in electrical engineering from the Instituto Tecno-

logico de Monterrey, Mexico, in 1974. From Stanford University, Stanford, California, he received in 1975 a MS in Electrical Engineering and a PhD in Computer Science in 1979. Garcia-Molina is a Fellow of the ACM, received the 1999 ACM SIGMOD Innovations Award, and is a member of the President's Information Technology Advisory Committee (PITAC).

Gio Wiederhold is a professor of Computer Science at Stanford University, with courtesy appointments in Medicine and Electrical Engineering. Since 1976 he has supervised 30 PhD theses in these departments. Current research includes privacy protection in collaborative settings, large-scale software composition, access to simulations to augment decision-making capabilities for information systems, and developing an algebra over ontologies.

Wiederhold has authored and coauthored more than 350 publications and reports on computing and medicine, including an early popular Database Design textbook. He initiated knowledgebase research through a white paper to DARPA in 1977, combining Databases and Artificial Intelligence technology. The results led eventually to the concept of mediator architectures.

Wiederhold was born in Italy, received a degree in Aeronautical Engineering in Holland in 1957 and a PhD in Medical Information Science from the University of California at San Francisco in 1976. Prior to his academic

career he spent 16 years in the software industry. He has been elected fellow of the ACMI, the IEEE, and the ACM. He spent 1991-1994 as the program manager for Knowledge-based Systems at DARPA in Washington DC. He has been an editor and editor-in-chief of several IEEE and ACM publications.