Searching Near-Replicas of Images via Clustering

Edward Chang, Chen Li, James Wang, Peter Mork and Gio Wiederhold
Department of Computer Science
Stanford University
{echang,chenli,wangz,pmork,gio}@cs.stanford.edu

Abstract

Internet piracy has been one of the major concerns for Web publishing. In this study we present a system, RIME, that we have prototyped for detecting unauthorized image copying on the World-Wide Web. To speed up the copy detection, RIME uses a new clustering/hashing approach that first clusters similar images on adjacent disk cylinders and then builds indexes to access the clusters made in this way. Searching for the replicas of an image often takes just one IO to look up the location of the cluster containing similar objects and one sequential file IO to read in this cluster. Our experimental results show that RIME can detect image copies both more efficiently and effectively than the traditional content-based image retrieval systems that use tree-like structures to index images. In addition, RIME copes well with image format conversion, resampling, requantization, and geometric transformations.
Keywords: clustering, copy detection, multidimensional indexes, similarity search.

1 Introduction

Advancement in the internet and World-Wide Web technology has led to the growth of information dissemination at an unprecedent pace. The resulting more and more efficient distribution of data, however, has increased the problems associated with copyright enforcement [11].

Many studies have proposed schemes for detecting copies of *text* documents [5, 22]. In this paper, we present an *image* copy detection service. We call this service RIME, for Replicated IMage dEtector. Given an image registered by its creator or distributor, RIME checks if *near-replicas* of the image (i.e., copies of the image that may have gone through altercations such as format conversion, resampling, requantization, and geometric transformations) exist on the internet and returns a list of *suspect* URLs. The creator or distributor of the image can then check if the suspect images are indeed unlawful copies (for instance by using watermarks).

Since images are characterized by high-dimensional feature vectors (RIME characterizes images using 192 wavelet coefficients), searching near-replicas of an image means essentially searching vectors in the neighborhood of the given image vector in high-dimensional spaces. A "brute-force" way to perform similarity queries is to read in all images, compute their distances to the query image, and select the closest images. However, the running time of this approach is proportional to the dimension (D) and size (N) of the dataset, or $O(ND)$, which can be very large. A logical way to improve upon this brute-force method is to reduce the search space. Many tree-like structures (e.g., R-trees, RStar-trees, TV-trees, SS-trees, SR-trees, X-trees, PK-trees, etc.) have been proposed to achieve search space reduction. However, recent studies [26, 28] point out that these tree-like structures suffer from the *curse of dimensionality* in that the time to traverse the

index structure to find the neighboring blocks (e.g., by using the branch-and-bound algorithm) and the number of neighboring blocks to check for similar images grow exponentially with the data dimension. Kleinberg [16] shows that, theoretically, using the brute-force algorithm provides a faster query time than using these indexing structures when $D \geq logN$. Worse yet, we show in Section 2 that when the data dimension is high, the traditional tree-like structures disperse the similar objects randomly on the disk. This random data placement together with the exponential growth of IOs makes these tree-like index structures very inefficient in performing similarity search in high-dimensional spaces.

In this paper, we present a new similarity search paradigm: a clustering/indexing combined scheme. Both clustering and indexing have been intensively researched, but these two subjects have been studied separately for different optimization objectives: clustering optimizes classification accuracy while indexing minimizes the storage requirement and the number of IOs. Because of these conflicting goals, the indexing schemes often do not preserve the clusters of the dataset and project objects that are close (hence similar) in high-dimensional spaces onto a 2D plane (the disk geometry) randomly. This is analogous to breaking a vase (cluster) apart to fit it into 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 is an astronomical job.

Given a large set of images, the data space is usually not uniformly occupied [21, 29]. Data clustering identifies the distribution patterns of the dataset and groups similar images into clusters. Therefore, we propose to store these clusters of similar images contiguously on disk and build indexes to locate them. Our clustering/indexing scheme first divides feature spaces into small cells like the longitudes and latitudes that divide a map into small sub-regions. These cells are used for clustering records that are similar and then for indexing the clusters. In the clustering phase, we tally the number of images residing in each cell and then group adjacent cells containing images into clusters. One can build a hierarchy of clusters by changing the resolution of the grid; that is, the finer the grid, the smaller the cell, and the finer the resulting clusters. (This clustering method is similar to using rectangles to approximate polynominal functions of any degree in calculus.) To improve IO efficiency, we store the finest grained clusters (we call them mountains) as individual sequential files and, as much as possible, the coarser-grained clusters (we call them islands, continents, etc.) in the same cylinder group. After the clusters are formed, the algorithm builds a mapping table for indexing the clusters. To answer a similarity query, we first hash the query image's feature vector to obtain its cell ID. Using the cell ID, we find the cluster where the query image resides by using the mapping table. The total number of IOs for a similarity query is usually two: one IO for the cluster lookup and one sequential file IO to read in the cluster. Intuitively, if an image has been replicated on the web, a cluster containing the image and its replicas is formed and the replicas can thus be found. If the query image is an outlier (i.e., no mountain contains the image), then the image may not have been copied. Therefore, this clustering/indexing scheme not only achieves high search efficiency, but also attains high copy detection accuracy.

Clustering data poses many challenges. We present the parameters that affect the size and the number of the clusters and outline the procedure to tune the parameters. Building indexes to support fast lookup in high-dimensional spaces and to conserve storage space at the same time is also a nontrivial task. We maintain a heap structure that keeps the size of the index structure linear with respect to the size of the database.

Summarize the contributions of this study:

- We propose the idea of mapping the data clusters onto storage clusters and building indexes to access the clusters made in this way.

- We present the algorithm to form the clusters and steps to adapt to the distribution of the dataset.
- We show that a similarity query can be conducted efficiently by the algorithm returning the cluster near the query image.
- We demonstrate that our index structure can support query refinement effectively.

We have built a Replicated IMage dEtector (RIME) prototype using the techniques presented in this paper to detect unlawful copy images. Our experimental results show that RIME can discover almost all suspect images (with format conversion, resampling, requantization, and geometric transformation) in the database. The prototype is available on the Web [1].

1.1 Related Work

Nearest-neighbor search techniques have been used to implement similarity search in high-dimensional spaces. However, due to the curse of dimensionality, some recent studies suggest doing similarity search only *approximately*. For example, White and Jain [28] suggest returning only the bucket of data in which the query object resides. This approach is fast but can suffer from very low recall and hence is not acceptable for detecting image replicas. Kleinberg [16] proposes two approximation algorithms that can run in $O((D\ log^2 D)(D + logN))$ and $O(N + D\ log^3 N)$ time. Two other approaches have also been proposed to deal with the curse of dimensionality: reducing the dimensionality and using parallel resources. These techniques make implementation feasible for some high-dimensional applications, but neither technique confronts the core issue directly.

At a first glance, image copy detection may be done by the traditional content-based image retrieval databases such as QBIC [10], Virage [13], VisualSEEk [23], and WISE [25], which retrieve images similar to a given one. However, these systems may suffer from the curse of dimensionality if they employed the traditional tree-like structures to search image replicas *precisely*, or they may suffer from poor recall if they employed the *approximate* search approach proposed in [28]. The clustering/indexing approach we propose in this paper avoids the curse of dimensionality by clustering similar images on disk. Although our clustering/indexing approach also performs similarity search approximately, it enjoys much higher recall since if replicas exist, a cluster is likely to form and thereby be detected.

Clustering techniques have been studied in statistics, machine learning, and database communities, all of which have focused on different aspects of the techniques. In the machine learning community, Self-Organizing Map (SOM) [17] and k-Mean are the most popular techniques developed for non-supervised training. SOM forms clusters given the size of a map and k-Mean forms clusters given the value of k (the number of clusters). But the best map size and the best value of k for a given dataset is difficult to learn. Consequently, these techniques may fragment what we call the natural clusters of the dataset. For instance, if the number of the natural clusters is not equal to k but we force the algorithm to generate k clusters, the resulting clusters may fragment the natural clusters. Indeed, our experimental results reported in [6] using Vector Quantization (VQ) [12] (a variant of k-Mean for high-dimensional data) show that VQ suffers from lower recall than that of the scheme proposed in this paper.

Recent clustering work in the database community includes CLARANS [19], BIRCH [29], DBSCAN [9], and CLIQUE [2]. These techniques have high degree of success in identifying clusters in a very large dataset, but they do not deal with *search and retrieval* efficiency. BIRCH, for example, uses a tree-like structure to form clusters. Since each leaf is limited to the page size, a cluster may have to be split to be stored on random parts of the disk.

Finally, many studies have proposed using watermark schemes to safeguard image copyright. These schemes add to the images the creator's or the distributor's identity. A watermark, however, is vulnerable to image processing, geometric distortions, and subterfuge attacks [8]. In addition, working with a large variety of watermark schemes increases the detection time drastically since for each internet image we may need to process it many times, each time with a different watermark scheme. Safeguarding copyright by checking the image directly has two distinct advantages: although a copied image can also be altered, it is unlikely that a pirate will change the image substantially and thereby lose its original appearance, and each image needs to be profiled only once. We believe that the watermark still plays an important role in authenticating images in courts of laws. After using our approach to provide the creator or distributor with a suspect list, the actual owner of the images can use watermark or other authentication techniques to prove ownership.

The rest of the paper is organized as follows: Section 2 shows the shortcomings of the traditional index structures in performing similarity queries in high-dimensional spaces. Section 3 presents our clustering/indexing algorithm. In Section 4, we show how a similarity query is conducted using our scheme. Section 5 briefly describes a prototype, RIME, that has been built for detecting image replicas on the Web and evaluates our clustering/indexing scheme both experimentally and analytically. Finally, we offer our conclusions in Section 6.

2 Traditional Index Structures

Many tree-like structures have been proposed to index high-dimensional data (e.g., R*-tree [3, 14], SS-tree [27], SR-tree [15], TV-tree [18], X-tree [4], M-tree [7], and K-D-B-tree [20]). In this section we discuss the shortcomings of using these tree-like structures in conducting similarity queries in high-dimensional spaces.

A tree-like structure divides the high-dimensional space into a number of sub-regions, 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 steps in most systems [24]:

1. It performs a *where-am-I* search to find out in which sub-region 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 Euclidean 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.[1] In the first step, if the index structure does not fit in the 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 sub-regions 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 $O(2^D)$ [24] Reading in the data from all neighboring regions can thus take an exponential number of IOs.

[1] To reduce search cost, some systems perform only the where-am-I lookup step and return all images in the same disk block where the query image is stored. This approach is not acceptable for applications like image copy detection [6] since having high recall is critical for the success of the system.

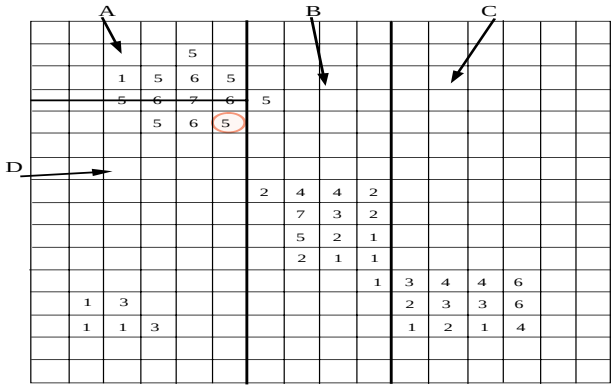

Figure 1: Clustering by Indexing

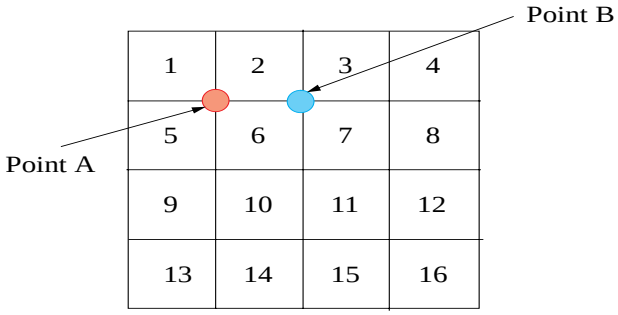

Figure 2: Random Placement Example

In addition to the large number of IOs, these IOs can be random and hence exceedingly expensive. An example can illustrate what we call the *random-placement* syndrome that the traditional index structures face. Figure 1 shows a 2D 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 sub-regions 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, left, middle, and right, and then to divide the left compartment horizontally. We are left with four sub-regions 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. This problem is analogous to having very short aisles in a library so that similar books must be shelved on different aisles. To find similar books, one must walk many aisles. For the index structure, the number of aisles (sub-regions) to look up for the neighboring points grows exponentially with respect to the data dimension. Furthermore, since in high-dimensional spaces the aisles cannot be arranged in a manner sequential to all possible query objects, the IOs must be random. Figure 2 shows a 2D example of this random phenomenon. The numbers in the figure represent the cell IDs. If a query point falls among cells 1, 2, 5, and 6 (point A in the figure), we wish these cells to be contiguous on disk so that the IO can be sequential. Suppose they are indeed contiguous on disk. Then a query point that falls among cells 2, 3, 6, and 7 (point B in the figure) must suffer from random IOs since it is impossible to keep these blocks, too, contiguous on disk. When the dimension is large, the neighboring blocks of a given object are scattered randomly on disk.

We summarize the shortcomings of the traditional structures for indexing in high-dimensional spaces as follows:

1. The search cost is high when the index structure cannot fit in the memory due to large D (number of dimensions) and large N (database size).
2. The cost of retrieving the blocks containing the neighboring objects can be very high because of the randomness and exponential growth of IOs.

To reduce the search cost, we propose a hashing approach. To overcome the random-placement syndrome we propose clustering and placing only similar objects in the same disk track or cylinder to make IOs sequential. We discuss our proposal in detail in Section 3.

3 Clustering for Indexing

In this section, we describe our clustering and indexing scheme. The main ideas of our scheme are to

- Cluster similar data on disk to minimize disk latency for retrieving similar objects, and
- Use hashing, rather than tree-like structures, to index clusters, thereby minimizing the cost of looking up a cluster.

Our approach consists of three steps:

1. We divide the i^{th} dimension into 2^κ stripes. In other words, at each dimension, we choose $2^\kappa - 1$ points to divide the dimension. (We describe how to adaptively choose these dividing points in Section 3.4.) This way, using a small number of bits (κ bits in each dimension) we can encode to which cell a feature vector belongs. Figure 3 shows a 2D sample grid structure that divides both x and y dimensions into 16 evenly divided stripes.
2. We group the cells into clusters. A cell is the smallest building block to construct clusters of different shapes. This is similar to the idea in calculus of using small rectangles to approximate polynominal 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 (details described in Section 3.1). Each cluster is stored as a sequential file on disk.
3. We build an index structure to refer to the clusters. A cell is the smallest addressing unit. A simple encoding scheme can *hash* an object to a cell id and retrieve the cluster it belongs to in one IO (details described in Section 3.2).

3.1 CF Clustering Algorithm

To perform efficient clustering in high-dimensional spaces, we propose the algorithm *cluster-forming* (CF). To illustrate the procedure, Figure 3 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.

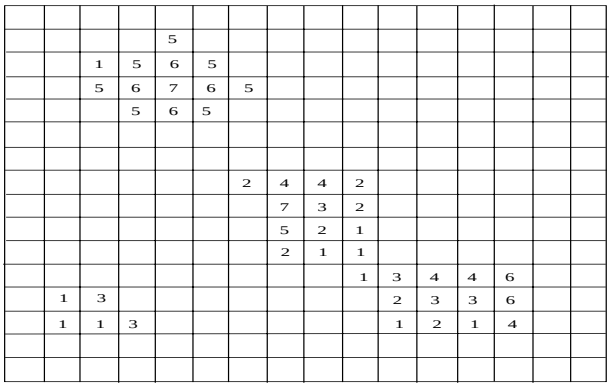

Figure 3: Grid Structure

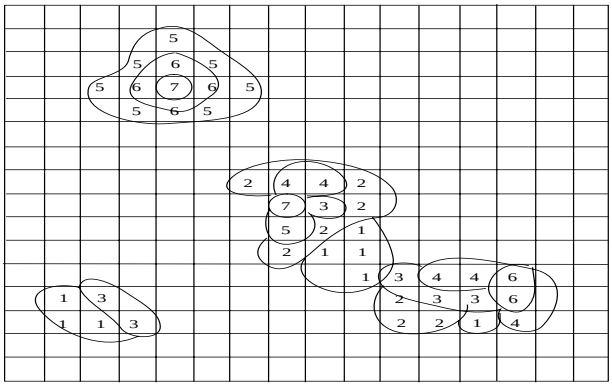

Figure 4: CF Example

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 3, we start with the cells marked 7.)

3. CF descends one unit at a time. 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 with the cluster.

   (c) If the cell is adjacent to more than one cluster, the CF algorithm invokes the *cliff-cutting* algorithm (CC) to determine to which cluster the cell belongs, or if the clusters should be combined.

4. CF terminates when the height drops to a threshold, which we call 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 4 shows the result of applying the CF algorithm to the data presented in Figure 3. In contrast to how the traditional indexing schemes split the data (shown in Figure 1), the clusters in Figure 4 follow what we call the natural clusters of the dataset.

The formal description of the CF algorithm is presented in Figure 5. The input to CF includes the dimension (D), the number of bits 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 ID ($C.id$), number of points in the cell ($C.\#p$), and the cluster 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 hashes the point into a cell ID. It then checks if 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). Both $HeapFind$ and $HeapInsert$ run in $O(lgN)$ time. Therefore, the running time of the first pass is thus $O(NlgN)$.

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 and then in steps 3 and 4 sorts the cells in the array in descending order on the point count ($C.\#p$). In the fifth step, 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 some existing clusters, 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 shortly.) In step 5.3, the cell joins the identified cluster (new or existing). Finally, the cells that are below the threshold are grouped into one cluster in steps 6 to 8 as outliers. Let $|H|$ denote the number of cells in the heap H. The running time of the second pass is $\Theta(|H|^2)$ because for each cell the algorithm needs to check the cells that are "higher" than that cell ($|H|/2$ cells on average). Notice that the value of $|H|$ depends only on how the N points are clustered and is independent from D. The storage requirement of CF is $O(ND)$, which we discuss this requirement in Section 3.2 in conjunction with discussion of the storage requirement for the object-to-cluster mapping table.

If a cell is adjacent to more than one cluster, the *cliff-cutting* (CC) algorithm determines to which cluster the cell belongs according to the following heuristics:

1. If the cell has a height only slightly below the peaks (e.g., one third from the peaks) of some adjacent clusters, CC combines these clusters.

2. If the neighboring cells of different clusters have the same altitude, CC joins the cluster with the smaller number of objects to make the points more evenly distributed.

3. Otherwise, CC connects the cell to a random adjacent cluster.

Many other heuristics exist. For instance, some boundary cells can be replicated in more than one cluster. Which rules are "better" depend on the dataset and the precision/recall requirements of the application.

3.2 Indexing the Clusters

After the clusters are formed, the final step is to build indexes to support fast access to them. Two structures are needed for book-keeping: a *mapping table* is maintained to map an object (represented by a vector) to a cluster and a *cluster directory* is built to record the information for the clusters.

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: $\tau \leftarrow Hash(p)$; /* hash the point into 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: $S \leftarrow \{C | C \in H\}$;

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: $\Psi \leftarrow FindNeighborClusters(S, C.id)$

5.2(a): if $(\Psi = \emptyset)$ /* not attach to any cluster */
new β; /* create a new cluster */
$\beta.C \leftarrow C.id$;
$\Phi \leftarrow \Phi \cup \{\beta\}$;

5.2(b): else If $(|\Psi| > 1)$
$\beta \leftarrow CC(C, \Psi, \Phi, H, S)$;

5.3: $C.\beta \leftarrow \beta$;

5.4: $i \leftarrow i + 1$;

6: new β; $\beta.C \leftarrow 0$;

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

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

Figure 5: The Cluster Forming (CF) Algorithm

The size of the mapping table is proportional to the size of the heap. The heap records the information about each cell. In the worst case, we have one cell for each point, so we can have at most N cell structures. Each cell structure records the cell ID, the point count, cluster ID, and some pointers for heap maintenance. The storage requirement for point count, cluster ID, and pointers is a small constant. The size of the cell ID is proportional to the data dimension D. The total storage requirement for the heap is thus on the order of $O(ND)$. Given an object in this heap, the time to look up a cluster is $O(lnN)$. In contrast to the tree-like index structures (shown in Section 2), our technique does not have a large memory requirement since we only read the block that contains the cell into memory when a search is conducted. The table's disk storage requirement is comparable to the tree-like index structures.

The second structure needed is a *cluster directory*. The cluster directory records for each cluster its ID, the name of the file in which the cluster stores the objects, and the cluster's centroid cell IDs. (The centroid cells are those with the highest point population.) Figure 4 in Section 4 shows an example of these data structures and how they are used.

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 parameter.

The number of cells is determined by parameters D and κ and can be written as $2^{D\times\kappa}$. The average population of a cell is thus $\frac{N}{2^{D\times\kappa}}$. Two factors decide the desired point density. On the one hand, we do not want to have low point density because it results in a large number of cells but a relatively small number of points in each cell. Having low point density tends to create many small clusters. On the other hand, we do not want to have densely populated cells either. Having high point density results in a small number of very large clusters that cannot help us to tell objects apart. Changing the value of κ affects the resolution of the grid and hence affects the number and the size of the clusters.

The value of θ also affects the number and the size of the clusters. Figure 6 shows an example in a one-dimensional space. The horizontal axis is the cell number and the vertical axis the number of points in the cells. The θ value set at the "A" level threshold forms three mountains (the shaded areas). The cells that do not belong to any mountain are clustered into a lake. Note that if we reduce the threshold, we may increase the mountain number or size, and decrease the size of the lake. We have two aims. On the one hand, we want the lake to be relatively small in terms of the number of points that fall into the lake to fit it into a few sequential disk blocks for IO efficiency. On the other hand, we want the lake to be relatively large in terms of the space it covers to keep the mountains well separated. The system should tune the values of κ and θ properly by considering the data distribution and the desirable cluster size.

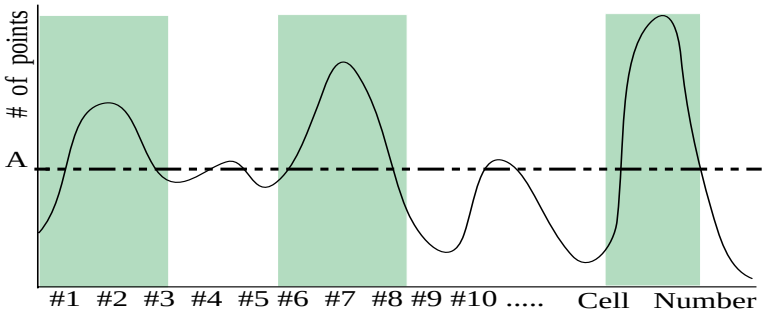

Figure 6: Threshold

3.4 Adaptive Clustering

Due to the uneven distribution of the objects, it is possible that some areas 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. For densely populated stripes, we can increase the quantization level. For instance, we can divide the regions with more points into smaller substripes. This way, we may avoid very large clusters, if such 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.

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 in traditional index-structure methods.

4 Similarity Search

Given a query object, we first map the object to a cluster using the mapping table. To illustrate, Figure 7 shows a simple 2D example.[2] Each cell is represented by a binary code starting from the x-axis and then the y-axis. For instance, the cell 0001 is at the intersection of the left-most stripe on the x-axis and the second stripe from the top on the y-axis. Figure 8 shows the index structures for the example. At the left side of the figure is the cell-to-cluster mapping table (we show the table uncompressed). Once the cluster is found, we use the cluster directory in the middle of the figure to find the sequential file where the cluster is stored.

In this example, the θ value is set to 3. The non-empty cells with a point population below 3 are thus clustered into a lake. (Note that empty cells do not take up space in the mapping table.) We have three clusters: two mountains and one lake. Mountain #1 consists of cells 0000 and 0100, and mountain #2 consists of cells 0110, 1010, 1110, and 1011. The cluster directory

[2]Our technique is most effective when the data dimension is high. However, we are constrained to use a low-dimensional example to illustrate our technique.

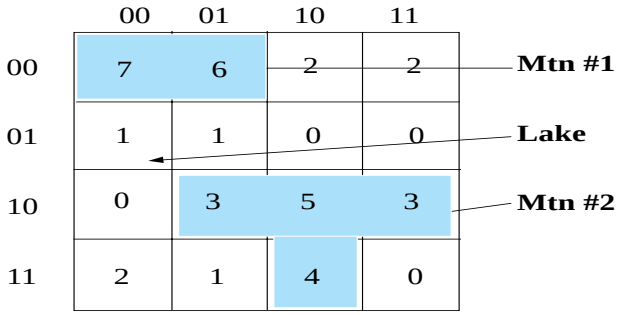

Figure 7: Cluster Example

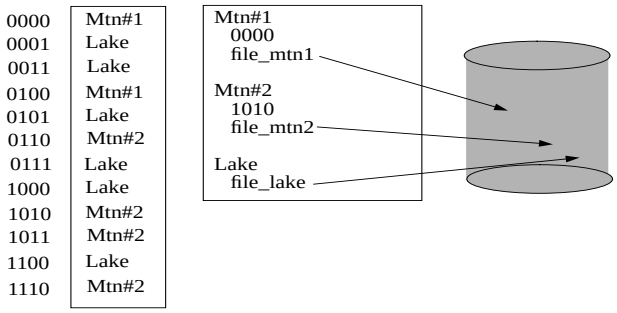

Figure 8: Index Structures

shows three clusters. The centroids of mountain #1 and #2 are cells 0000 and 1010, respectively. The names of the files storing the clusters are also recorded in the directory.

Given a query object, we first quantize its x and y attributes into a four-bit code. We then look up in the mapping table on which cluster the object resides. If the query object falls on a mountain, we read in the disk blocks belonging to the mountain sequentially from the disk, rank the points using some distance functions, then present the result to the user. For instance, if the query object is hashed into the 0000 cell, we return those 13 objects on mountain #1 in some sorted order.

4.1 Query Refinement

Supporting query refinement is important for similarity queries. Users may not at all know what they are looking for in the first place, so the queries must be interactive. If the query object belongs to a mountain cluster, returning the mountain is likely to satisfy the user. If the query object does not belong to any mountain cluster, then the user needs to have a way to progressively move toward his/her desired result.

When the query object does not belong to a mountain, it falls into a lake. For example, a query object belonging to cell 0001 falls into the lake (shown in Figure 7). Although a lake may contain objects from far-away cells, some distance functions can help filter out those objects. At the same time, we can obtain samples from the near-by mountains by retrieving their centroid cells (the objects in the centroid are stored near the beginning of the cluster's file). For instance, if the query object falls into cell 0001, we return all points in the lake and the points in cells

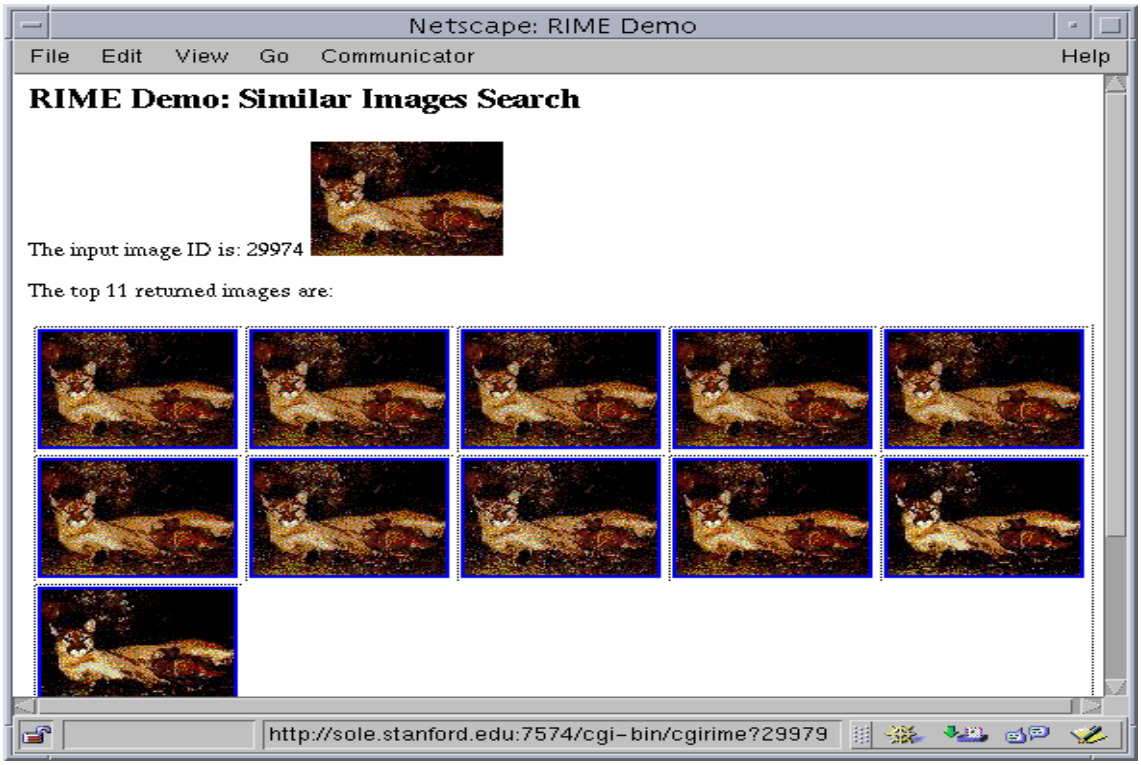

Figure 9: The Results of Querying Image #29974

0000 (centroid of mountain #1) and 1010 (centroid of mountain #2). If the user picks an object residing on, say, mountain #1, then the subsequent refinement returns all points belonging to this mountain.

5 Evaluation

We evaluate the effectiveness of our clustering/indexing scheme in two parts. First, we show some experimental results using the prototype we have built to detect image near-copies. Second, through analysis we show the reasons why our scheme can detect image replicas both efficiently and effectively.

5.1 Experimental Results

We built an image database with about $30,000$ images. For each image we prepared its feature vector in three steps. First, we converted the image into a canonical size and format. Second, we applied Daubechies' wavelet transformation on the image to produce coefficients. Finally, we stored the selected 8×8 wavelet coefficients in three color spaces, C_1, C_2, and C_3, respectively as the image's feature vector. For the detailed steps of feature vector preparation, please refer to [6].

To cluster the dataset properly we tuned parameters κ and θ iteratively to calibrate the clustering algorithm and thereby to find the patterns in the dataset. We experimented with

different values of κ and θ to adjust the cluster size and cluster number. When κ was set to two (four stripes per dimension) and θ to one, we obtained 215 clusters with an average size of 139 images. Most similar images fall into the same cluster, which shows that our method can be more accurate than the the vector quantization method reported in [6] to index the same set of images.

For testing the accuracy, we inserted a sequence of 10 images modified from the same image to the database of 30,000 images. The image was processed through sharpening, softening, despeckling, posterizing, and watercoloring, and requantization. Figure 9 shows a sample query result obtained from the system. The image at the top and at the upper-left corner of the query result in the figure is the query image. We were able to detect all ten near-replicas. This result is better than our previous attempt using the VQ technique to index the feature vectors, where only 80% of the near-replicas were detected. Due to the limited space, we cannot report all our experimental results. However, our prototype is available on-line [1] for the readers to conduct verification and further experiments.

We close our experiment report with two notes. First, in addition to the accuracy, RIME returns query results (including locating the cluster, reading in the cluster from disk, computing and ranking the results, and returning the top 20 similar images) very fast, in about two seconds. However, if the test is run at a remote site with a slow network connection, the transmission delay of the results may take a few more seconds. Second, we index the images using only the color features. We are currently adding shape and texture information to the feature vectors to enhance the precision (and hence further reduces the IO cost) of the system.

5.2 Analysis

Let p denote the probability that the user is able to find the desired result in the first attempt, T_{read} the IO time to read in the final desired cluster, and T_{refine} the IO time spent in the query refinement stage. The search time T_{search} can be modeled as

$$T_{search} = (1 - p)T_{refine} + T_{read}. \tag{1}$$

To minimize T_{search}, we want to maximize p as well as minimize the IO time. The value of p is the probability that the query object is placed correctly into the cluster it belongs to. The cost of the search is on the order of one sequential file IO, and it depends on the average size of the clusters.

5.2.1 Clustering Accuracy

When a query image is near the middle of a cluster, there is little chance that we can miss detecting the near-replicas of the image. Mis-detection can happen if the query image is around the rim of a mountain or is in a lake. In these cases, some of the query image's adjacent points may be classified into some neighboring clusters and can escape from being detected. However, if an image has been replicated, the cell where the query image is in and its neighboring cells are likely to contain enough points to form a cluster. In other words, the more an image is replicated, the more likely the clustering scheme can find the near-replicas.

Using the tree-like structures or VQ to detect replicas approximately, on the other hand, can miss-detecting a large number of replicas, regardless if the query image has been copied extensively. The mis-detection results from that these techniques often fragment a natural cluster,

Parameter Name	*Value*
Disk Capacity	*9.0 GBytes*
Avg. Transfer Rate TR	*15 MBytes*
Avg. Rotational Latency Time	*4.17 milliseconds*
Min. Seek Time	*1.5 millisecond*
Avg. Seek Time	*7.1 milliseconds*

Table 1: Seagate Barracuda 9L Disk Parameters

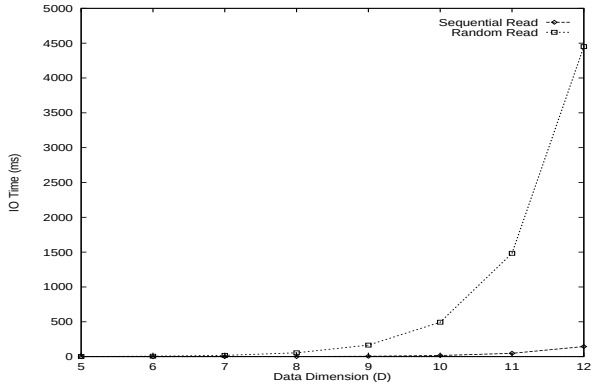

Figure 10: Sequential vs. Random IO Performance

as we have illustrated in Figure 1. Worse yet, the more an image is replicated, the more near-replicas of the image can be mis-detected, if these techniques return only the points in the bucket where the query image resides.

5.2.2 Sequential versus Random IOs

Using our cluster/indexing scheme tends to read in more data than that of using the traditional tree-like structures (i.e., lower precision). In this section we show that since the IOs performed by our scheme are sequential, the performance is much more likely to be better. Moreover, reading in more data helps increasing recall, which is critical for an image copy detector. The copy detector using our scheme may have more points to evaluate after they are read into main memory. But since IO, not in-memory distance computation, is the performance bottleneck of the search, being able to read in more data in a sequential manner is far more important than reading in less data in a random manner, as we illustrate next.

For a sequential read, the disk pays an average seek and an average rotational latency and then the time to transfer S amount of data. Let $\gamma(d)$ denote the seek time function, which computes seek time given seek distance d, T_{rot} denote the average rotational delay, and S/TR denote the transfer time for S amount of data at transfer rate TR. A sequential IO time can be modeled as

$$T_{read} = \gamma(d) + T_{rot} + S/TR.$$

For a random read (by which we mean that the blocks belonging to S are scattered randomly on the disk), the read time is the number of blocks to transfer (S/B, where B is the block size)

times the IO time to seek and then transfer a block, or

$$T_{read} = S/B(\gamma(d) + T_{rot}) + S/TR.$$

If an elevator-like disk scheduling policy is employed, the seek distance for a random read can be amortized among the blocks the read retrieves. Assume that two sweeps of the disk find all the needed blocks (the average seek distance is $2 \times CYL$ divided by S/B).

Using the disk parameters listed in Table 1, we compare the performance of sequential versus random IOs under different data dimensions D in Figure 10. The x-axis of the figure shows the dimension of the data, from $D = 5$ to 12. The y-axis shows the IO time from retrieving 3^D number of blocks. It is obvious that the random IOs are much more expensive than the sequential IOs. For instance, the time to read in $1,000$ random disk blocks can be used to read in $28,000$ sequential blocks. The time to read in $10,000$ random blocks ($D = 9$) can be used to read in more than $300,000$ sequential blocks. It is also obvious that even a sequential file may contain points that are not close to the query object (i.e., lower precision), our clustering approach is very likely to perform much better than the traditional index structures.

6 Conclusions

In this paper, we presented a new paradigm for performing similarity search for high-dimensional data. Instead of translating a similarity search into a range search, we cluster the data and navigate the search among clusters. This approach improves the IO efficiency by clustering and retrieving relevant information sequentially on and from the disk. It also provides a way to incorporate a user's relevant feedback to perform query refinement by moving the subsequent search towards a more relevant cluster. We have built a Replicated IMagE Copy Detector (RIME) using this clustering algorithm. The system enjoys both high copy-detection accuracy and efficiency.

We are currently crawling more internet images to build a much larger database to test out the robustness of the feature vector and index structure.

References

[1] http://www-db.stanford.edu/~echang/papers/rime.html.

[2] 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.

[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] S. Berchtold. The x-tree: An index structure for high-dimensional data. *Proceedings of the 22nd VLDB*, August 1996.

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

[6] E. Chang, J. Wang, C. Li, and G. Wilderhold. Rime - a replicated image detector for the world-wide web. *Proceedings of SPIE Symposium of Voice, Video, and Data Communications*, November 1998.

[7] 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.

[8] I. Cox, J. Kilian, F. Thomson, and T. Shamoon. Secure spread spectrum watermarking for multimedia. *IEEE Transaction on Image Processing*, 6(12):1673–86, December 1997.

[9] 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.

[10] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, and et al. Query by image and video content: the qbic system. *IEEE Computer*, 28(9):23–32, 1995.

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

[12] A. Gersho and R. Gray. *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, 1992.

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

[14] A. Guttman. R-trees: a dynamic index structure for spatial searching. *Proceedings of ACM Sigmod*, June 1984.

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

[16] J. M. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. *Proc. of 29th ACM Symposium on Theory of Computing*, 1997.

[17] T. Kohonen. *Sel-Organizing Maps*. Springer, Berlin, Heidelberg (Second, extended edition), 1997.

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

[19] R. T. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. *Proceedings of the 20th VLDB*, September 1994.

[20] J. T. Robinson. The k-d-b-tree: A search structure for large multidimensional dynamic indexes. *Proceedings of ACM SIGMOD*, April 1981.

[21] 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.

[22] N. Shivakumar and H. Garcia-Molina. Finding near-replicas of documents on the web. *Proceedings of Workshop on Web Databases (WebDB'98)*, March 1998.

[23] J. R. Smith and S.-F. Chang. Visualseek: A fully automated content-based image query system. *ACM Multimedia Conference*, 1996.

[24] J. Ullman, H. Garcia-Molina, and J. Widom. Database system principles lecture notes. 1998.

[25] J. Z. Wang, J. Li, and G. Wiederhold. Wise: A wavelet-based image search engine with efficient feature vector clustering and classification. *Submitted for journal publication*, 1998.

[26] 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.

[27] D. A. White and R. Jain. Similarity indexing with the ss-tree. *Proceedings of the 12th ICDE*, Feb. 1996.

[28] D. A. White and R. Jain. Algorithms and strategies for similarity retrieval. *Stanford Technical Report*, August 1998.

[29] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: An efficient data clustering method for very large databases. *Proceedings of Sigmod*, June 1996.