The GC-Tree: A High-Dimensional Index Structure for Similarity Search in Image Databases

Guang-Ho Cha and Chin-Wan Chung

***Abstract*—With the proliferation of multimedia data, there is an increasing need to support the indexing and retrieval of high-dimensional image data. In this paper, we propose a new dynamic index structure called the *GC-tree* (or the *grid cell tree*) for efficient similarity search in image databases. The GC-tree is based on a special subspace partitioning strategy which is optimized for a clustered high-dimensional image dataset. The basic ideas are threefold: 1) we adaptively partition the data space based on a *density function* that identifies dense and sparse regions in a data space; 2) we concentrate the partition on the dense regions, and the objects in the sparse regions of a certain partition level are treated as if they lie within a single region; and 3) we dynamically construct an index structure that corresponds to the space partition hierarchy. The resultant index structure adapts well to the strongly clustered distribution of high-dimensional image datasets. To demonstrate the practical effectiveness of the GC-tree, we experimentally compared the GC-tree with the IQ-tree, the LPC-file, the VA-file, and the linear scan. The result of our experiments shows that the GC-tree outperforms all other methods.**

***Index Terms*—Dynamic index structure, GC-tree, high-dimensional indexing, image database, nearest neighbor search (NN search), similarity search.**

I. Introduction

SIMILARITY search in high-dimensional image databases is an interesting and important, but difficult problem. The most typical type of similarity search is the k-nearest neighbor (k-NN) search. The traditional k-NN problem is defined as follows. Consider a database *DB* consisting of points from $S = R_1 \times \cdots \times R_d$, where $R_i \subseteq \Re$. Each R_i usually consists of either integers or floats. A k-NN query consists of a point $\boldsymbol{q} \in S$ and a positive integer k. The k-NN search finds the nearest k neighbors of $\boldsymbol{q}$ with respect to a distance function $||\cdot||$. The output set O consists of k points from the database such that

$$\forall \boldsymbol{a} \in O \text{ and } \forall \boldsymbol{b} \in DB - O \ ||\boldsymbol{q} - \boldsymbol{a}|| \leq ||\boldsymbol{q} - \boldsymbol{b}||.$$

The actual problem in image database applications is how to process such queries so that the nearest k objects can be returned within the desired response time. Therefore, our focus is the development of an indexing method to accelerate the speed of the k-NN search.

For applications where the vectors have low or medium dimensionalities (e.g., less than 10), the state-of-the-art tree-based indexing techniques such as the R*-tree [2], the X-tree [5], the HG-tree [7], and the SR-tree [14] can be usefully employed to solve the k-NN problem. So far, however, there is no effective solution to this problem for the applications in which the vectors have high dimensionalities, say over 100. Therefore, the main issue is to overcome the *curse of dimensionality* [20]–a phenomenon that the performance of indexing methods degrades drastically as the dimensionality increases.

A. Motivation

Recently, we developed a new vector approximation-based indexing method called the *local polar coordinate (LPC)-file* [6] for the k-NN search. The LPC-file significantly improved the search performance for large collections of high-dimensional vectors compared with the linear scan and the VA-file [23]. The linear scan is often used as the yardstick for comparing with other indexing methods since most tree-structured indexing methods could not defeat it in high-dimensional data spaces. The VA-file was the first vector approximation-based indexing method to overcome the dimensionality curse. Although the LPC-file provided significant improvements compared with previous techniques, it suffers the performance degradation if the dataset is highly clustered because it employs a simple space partitioning strategy and the uniform bit allocation strategy for representing the partitioned region.

In the current vector approximation approach including the VA-file and the LPC-file, there is an implicit assumption that it is very unlikely that several points lie in the same cell. Actually, the vector approximation approach benefits from the *sparseness* of the high-dimensional data space as opposed to the partitioning or clustering-based approach (i.e., the traditional indexing methods) when several data points are assumed not to fall into the same cell. However, if the data points are highly clustered as in image datasets, the probability that a certain cell includes several points increases, and therefore those vectors may use the same approximation. This means that the discriminatory power of the approximation decreases and thus less elimination of candidates is performed in each phase of the k-NN search, and ultimately more disk accesses are required during the search.

Figs. 1 and 2 show the vector selectivity comparison between random and real image datasets during the first filtering and the second refinement phases of the k-NN search in the vector approximation approach, respectively. The *vector selectivity* is de-

Manuscript received April 21, 2001; revised February 26, 2002. This work was supported by the University Fundamental Research Program of Ministry of Information and Communication in the Republic of Korea under Grant 2001-116-3. The associate editor coordinating the review of this paper and approving it for publication was Dr. Sankar Basu.

G.-H. Cha is with the Department of Multimedia Science, Sookmyung Women's University, Seoul 140-742, South Korea (e-mail: ghcha@sookmyung.ac.kr).

C.-W. Chung is with the Department of Computer Science, Korea Advanced Institute of Science and Technology, Taejon 305-701, South Korea (e-mail: chungcw@islab.kaist.ac.kr).

Publisher Item Identifier S 1520-9210(02)04857-5.

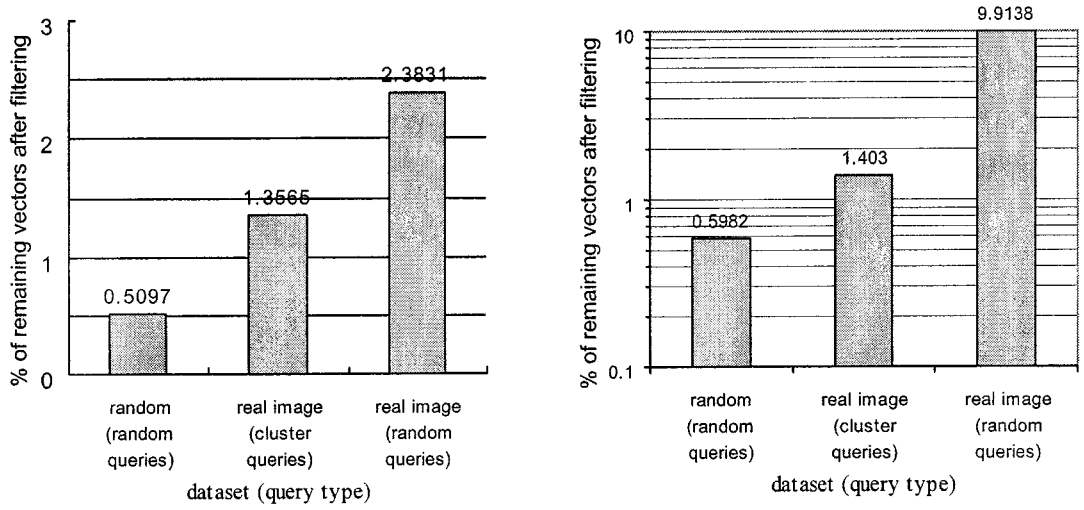

Fig. 1. Selectivity comparison between random and real image datasets in the filtering phase. (a) LPC-file. (b) VA-file.

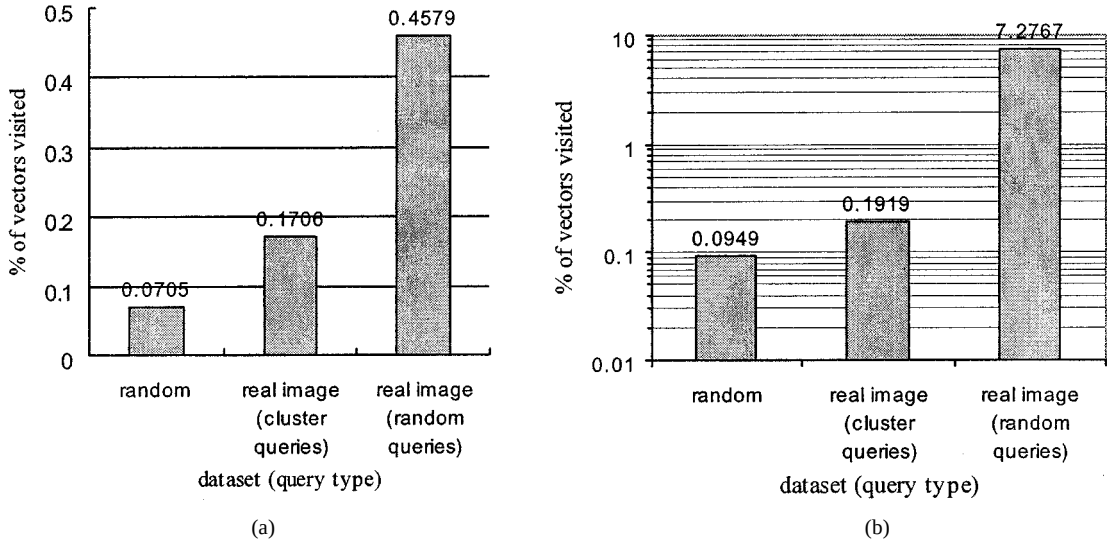

Fig. 2. Selectivity comparison between random and real image datasets in the refinement phase. (a) LPC-file. (b) VA-file.

fined as the ratio of the number of vectors visited to the total number of vectors in a dataset. The vector selectivity is a good performance estimator because the performance of the k-NN search depends largely on the number of disk blocks visited and it is affected by the vector selectivity. The selectivity of the first search phase of the vector approximation approach means the ratio of candidates not eliminated after the first filtering phase to the total number of vectors. The selectivity of the second phase is the ratio of real vectors visited during the second phase to the total number of vectors. In Fig. 1, the x-axis represents the datasets and query types used in our experiment and the y-axis represents the percentage of remaining vectors after the first filtering phase (i.e., the vector selectivity of the first phase). The cluster query means that the query vector is selected from vectors in the real image dataset itself, and the random query means that the query vector is selected randomly. In the experiment for the vector selectivity comparison, we used 256-dimensional image and random datasets. For the random dataset, 1000 random 10-NN queries were posed, and for the real image dataset, 1000 random 10-NN queries and 1000 cluster 10-NN queries were posed, and their results were averaged. In the case of the LPC-file in Fig. 1(a), the experimental result shows that, after the first phase filtering, the number of remaining vectors for a real image dataset is 2.66 to 4.68 times more than that for a random dataset. Similarly, in the case of the VA-file in Fig. 1(b), after the first phase filtering, the number of remaining vectors for a real image dataset is 2.34 to 16.57 times more than that for a random dataset. These mean that, compared with the random dataset, for the clustered dataset, the filtering power of the current vector approximation approach decreases drastically.

For the second refinement phase, in the case of the LPC-file as shown in Fig. 2, the number of real vectors visited for a real image dataset is 2.43 to 6.49 times more than that for a random dataset. In the VA-file, the number of real vectors visited for a real image dataset is 2.02 to 76.67 times more than that for a random dataset. These mean that the I/O cost occurred during the k-NN search for the clustered dataset is much higher than that for the random dataset.

This performance degeneration for the clustered dataset is a common problem of the current vector approximation approach.

TABLE I
SUMMARY OF SYMBOLS AND DEFINITIONS

Symbols	Definitions
d	number of dimensions
N	number of data points in a database
k	number of nearest neighbors to find
$\boldsymbol{p}$	database point
$\boldsymbol{q}$	query point
c	cell vector representing the region inside which $\boldsymbol{p}$ lies
$k\text{-}NN^{dist}$	distance between $\boldsymbol{q}$ and the k-th nearest neighbor of $\boldsymbol{q}$
$k\text{-}NN^{sphere}$	sphere with center $\boldsymbol{q}$ and radius $k\text{-}NN^{dist}$

The reason for the performance degeneration is because bits are uniformly allocated to each partitioned cell while the data distribution is not uniform depending on the cells and the density of each cell varies greatly. The way to overcome this problem is to adaptively assign bits to each cell according to the data statistics of the cell. This is one of the design objectives of the GC-tree.

Another problem of all vector approximation techniques is that they have to sequentially read the whole approximation file. To overcome this limitation of the vector approximation approach and to take advantage of the multidimensional index structure, we combine their advantages. In order to achieve this goal, we partition the data space based on the *density* of subspaces and construct an index structure that reflects the partition hierarchy of the data space. Compared with the vector approximation techniques, it can considerably reduce the number of disk accesses during the search since it narrows the search space based on the index instead of scanning the whole approximation file.

B. Testbed, Assumptions, and Notations

The testbed for our exploration to the k-NN problem is IBM's query by image content (QBIC) system [9], [20]. We focus on the specific problem of retrieving images via a 256-dimensional color histogram, using a QBIC's special-purpose color-similarity measure. In this paper, we assume that the domain space is the d-dimensional Euclidean space where the dissimilarity is measured by the Euclidean distance weighted by the matrix A, where A is a symmetric color similarity matrix [20]. In addition, users are assumed to pose a k-NN query using a sample image stored in the database.

Table I gives the summary of symbols and their definitions used in the paper.

C. Contribution of the Paper

We present a dynamic index structure, the GC-tree, that combines the capability of the vector approximation technique that accesses only a small fraction of real vectors with the advantage of the multidimensional index structure that prunes most of the search space and constructs the index dynamically. The GC-tree partitions the data space based on the analysis of the data distribution and identifies subspaces with high density. It focuses the partitioning on those dense subspaces and dynamically constructs an index that reflects the partition hierarchy. We provide algorithms for k-NN search as well as for the index construction. An empirical evaluation shows that the GC-tree significantly improves the performance of the vector approximation techniques for the real image database.

The organization of the paper is as follows. We begin by discussing the related work presented in the literature in Section II. Sections III and IV are the heart of the paper where we present the GC-tree and its algorithms. We present a performance evaluation in Section V, and conclude with summary and future work in Section VI.

II. RELATED WORK

In the recent literature, a variety of indexing methods suitable for high-dimensional data spaces have been proposed. Most of the work was motivated by the limitation of the state-of-the-art tree-structured indexing methods. We classify them into six categories:

1) dimensionality reduction (DR) approach [8], [13], [17];
2) approximate nearest neighbor (ANN) approach [1], [12], [15];
3) multiple space-filling curves approach [18], [22];
4) vector approximation (VA) approach [6], [23];
5) hybrid approach [3];
6) nearest neighbor (NN) cell approach [4].

The DR approach first condenses most of information in a dataset to a few dimensions by applying the singular value decomposition (SVD), the discrete cosine transform (DCT), or the discrete wavelet transform (DWT). The data in the few condensed dimensions are then indexed using a multidimensional index structure. While the DR approach provides a solution to the dimensionality curse, it has several drawbacks [6]: 1) loss of precision of the query result; 2) static method; and 3) dimensionality is still high even after reduction. However, the DR approach is one of the solutions to improve the search performance in high-dimensional space if the loss of information is acceptable within a certain range.

The idea behind the ANN approach is to retrieve k ANNs faster within a given error bound ϵ instead of retrieving exact k NNs. Given a query point $\boldsymbol{q}$ and a distance error $\epsilon > 0$, a point $\boldsymbol{p}$ is a $(1+\epsilon)$-*ANN* of $\boldsymbol{q}$ such that for any other database point $\boldsymbol{p'}$, $\|\boldsymbol{q}-\boldsymbol{p}\| \leq (1+\epsilon)\|\boldsymbol{q}-\boldsymbol{p'}\|$.

The multiple space-filling curves approach orders the d-dimensional space in many ways, with a set of space-filling curves such as Hilbert curves, each constituting a mapping from $R^d \rightarrow R^1$. This mapping gives a linear ordering of all points in the data set. Therefore, when a query point is mapped to the space-filling curve, one can perform a range search for nearby points along the curve to find near neighbors in the data space. However, due to the nature of the $R^d \rightarrow R^1$ mapping, some near neighbors of the query point may be mapped far apart along a single curve. To make sure that these points are not overlooked, multiple space-filling curves are used, based on different mappings from $R^d \rightarrow R^1$. Although this approach improves the performance of the k-NN search, it is possible that some near-neighbors may be omitted during the search. To improve the chances of finding all k NNs, more space-filling curves have to be used

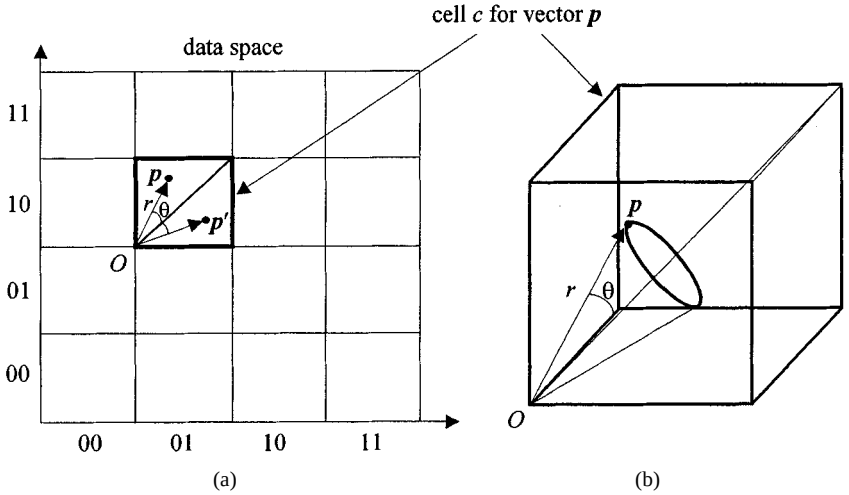

Fig. 3. Vector p and its approximation in the LPC-file. (a) Two-dimensional space. (b) Three-dimensional space.

or a larger neighborhood on each curve has to be scanned. However, they diminish the benefits of the approximate approach since both of these methods increase the size of the candidate set.

The VA approach tries to overcome the dimensionality curse by filtering the feature vectors so that only a small fraction of them must be visited during the search. The VA-file [23] and the LPC-file [6] are classified into this category. We discuss the VA approach somewhat in detail because the GC-tree combines this approach with the multidimensional index structure.

The idea of the VA approach starts from the observation that existing multidimensional structures cannot defeat the linear scan in high dimensions. Therefore the VA approach constructs the approximation file for the original data, and during the search, it sequentially reads the relatively smaller approximation file instead of the data file and tries to filter the original vectors so that only a small fraction of them must be read.

The VA-file divides the data space into 2^b rectangular cells, allocates a unique bit-string of length b to each cell, and approximates the data points that fall into a cell by that bit-string. The VA-file itself is simply an array of these bit-string approximations of data points. The performance of the VA approach heavily depends on the quality of the approximation since the filtering is performed based on the approximation.

The LPC-file aims at improving the filtering power of the vector approximation with a minimum amount of additional information. It enhances the filtering power of the approximation by adding polar coordinate information of the vector to the approximation. The LPC-file partitions the vector space into hyper-rectangular cells and approximates vectors using the *polar coordinates* on the partitioned local cells.

In the LPC-file, the approximation $\boldsymbol{a}$ for a vector $\boldsymbol{p}$ is generated as follows. The first step is to assign the same number of bits b to each dimension and to divide the whole data space into 2^{bd} cells. Thus the lengths of sides of a cell are the same. The cell is simply represented by the concatenation of the bit patterns for each dimension in turn. Fig. 3(a) shows an example in a two-dimensional (2-D) space: the cell c is represented by the sequence of bits (01, 10) where $d = 2$ and $b = 2$.

The second step is to represent the vector $\boldsymbol{p}$ using the polar coordinates (r, θ) within the cell c in which $\boldsymbol{p}$ lies. As illustrated in Fig. 3(a), the local origin O of each cell is determined as the lower left corner of the cell. The distance r of the polar coordinate is computed by the distance between the local origin O and the vector $\boldsymbol{p}$. The angle θ is computed by the angle between the vector $\boldsymbol{p}$ and the diagonal from the local origin to the opposite corner. As a result of this approximation, the vector $\boldsymbol{p}$ is represented by the triplet $\boldsymbol{a} = \langle c, r, \theta \rangle$, where c, r, and θ denote, respectively, the approximation cell, the distance, and the angle of $\boldsymbol{p}$ based on the local origin O.

One of the drawbacks of the VA techniques is that they have to read the whole approximation file for filtering. This is inevitable for the VA approach to filter the real data. Thus, the performance of the VA approach largely depends on the size of the approximation file and the filtering capability of the approximation. One of the motivations of the design of the GC-tree is to reduce the amount of approximations that have to be read during the search by employing the index structure as well as to maintain the index structure dynamically.

The design objective of the IQ-tree [3] is similar to that of the GC-tree in the sense that it employs the hybrid approach that combines the VA approach and the tree-based indexing. The IQ-tree has a three-level structure: The first level is a flat directory consisting of minimum bounding rectangles, the second level contains the approximations and the third level contains real points. Although the design objective of the IQ-tree is similar to that of the GC-tree, its approach is very different from that of the GC-tree. While the GC-tree partitions a space into 2^d cells at a time to identify clusters and outliers based on the local density of subspaces, the IQ-tree bisects a d-dimensional space using a $(d-1)$-dimensional hyper-plane. In addition, while the GC-tree maintains the hierarchical directory corresponding to the partition hierarchy, the IQ-tree uses a flat directory.

The NN-cell approach [4] precomputes the result of any k-NN search which corresponds to a computation of the Voronoi cell of each data point, and stores the Voronoi cells in an index tree. Then the k-NN search corresponds to a point query on the index tree.

III. The GC-Tree

The research challenge which has led to the design of the GC-tree is to combine the capability of the vector approximation approach that accesses only a small fraction of real vectors with the advantage of the multidimensional index structure that prunes most of the search space and constructs the index dynamically. In order to achieve this goal, we partition the data space based on the analysis of the dataset and construct the hierarchical index that reflects the space partition hierarchy.

A. Density-Based Space Partitioning

The GC-tree employs a *density-based* approach to partition the data space and to determine the number of bits to represent a cell vector for a partition. To approximate the density of the data points, the GC-tree partitions the data space into nonoverlapping hyper-square *cells* and finds the points that lie inside each cell of the partition. This is accomplished by partitioning every dimension into the same number of equal length intervals at a time. This means that every cell generated from the partition of a space has the same volume, and therefore the number of points inside a cell can be used to approximate the density of the cell.

In a static database, the *density* of a cell can be defined as the fraction of data points in the cell to the total data points. However, for a dynamic database environment, and especially, in the case of constructing a database from scratch, it is difficult to estimate the *density threshold* that identifies the dense and sparse cells because the density is relatively determined with respect to the total data points. Therefore, in the GC-tree, we define the *density* of a cell to be the proportion of data points in the cell to disk page capacity when we divide a space into 2^d cells by binary partitioning. A (sub)space in the data space corresponds to a *node* in the GC-tree and is physically mapped to a single *disk page*. There is a number P that identifies the maximum number of objects that can be accommodated in a disk page. That is, P represents the *page capacity* or the *fanout* of a page. When the number of objects inserted into a page exceeds P, the page is generally split into two. We call a cell c *dense* if the density of c is greater than or equal to a certain *density threshold* τ. Otherwise, it is called *sparse*. We call a dense cell a *cluster* and call the points that lie inside sparse cells *outliers*. If we determine the density threshold τ to be larger than a half of the page capacity, at most one cluster can be generated when we partition a space due to the insertion of an object.

The basic idea of the density-based partitioning is: 1) to identify clusters and outliers when we partition a space; 2) to focus the partitioning on the subspaces of the clusters found because the subspaces covered by the outliers are unlikely to be pruned in the search; and 3) to deal together with all outliers found in the partitioning of a certain space.

It is difficult to bind the outliers within a small region since they are widely spread over the whole subspace. Thus it is very difficult to prune the outliers collectively during the search because the large $k\text{-}NN^{\text{sphere}}$ is likely to intersect the large bounding region in which the outliers lie. Therefore, we collect in a single node of the GC-tree all outliers generated from a single subspace partition, and concentrate the partitioning on the clusters to reduce the possibility that clusters are intersected by the search sphere $k\text{-}NN^{\text{sphere}}$. If the number of outliers generated from the partition exceeds the page capacity, the GC-tree allocates more pages for the outliers and simply links them. It makes multiple pages a single *virtual page*. This is based on the observation that the volume covered by the outliers is so large that it may not be pruned in the search.

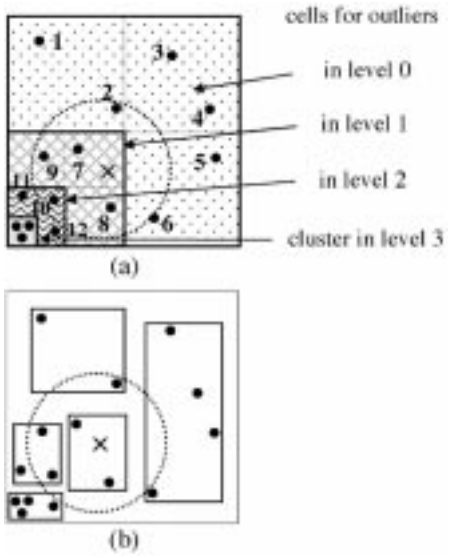

Fig. 4. Partitioning strategies. (a) GC-tree partitioning. (b) Traditional partitioning.

It is well known that for low-dimensional indexes, it is beneficial to partition the data space as *balanced* as possible. However, in high-dimensional spaces, the balanced partitioning results in large bounding rectangles for the partitions. When we apply balanced partitioning on a uniformly distributed dataset, the data space cannot be split in each dimension. For example, in a 256-dimensoinal data space, a split in each dimension results in a 2^{256} partitions (or disk pages). Therefore, the data space is usually split once in a number d' of dimensions. In the remaining $(d - d')$ dimensions it has not been split and the bounding rectangles include almost the whole data space in these dimensions. Even for the nonuniformly distributed (e.g., clustered) dataset, the bounding rectangles are likely to be large because they still try to accommodate outliers and the outliers usually lie far apart. On the contrary, the GC-tree excludes the outliers in forming the bounding regions to reduce the size of the bounding regions.

Fig. 4 depicts the partitions resulted from the density-based partitioning of the GC-tree and the traditional balanced partitioning in a 2-D example. In the GC-tree, a subspace is partitioned into 2^d cells at a time by splitting each dimension of the subspace in its center. In high dimensions, in fact, the vast majority of cells of the partitioned 2^d cells must be empty since 2^d is much larger than the number of objects in a database when d is large, say over 100. Thus the outliers are far apart one another and the size of the cell that includes clustered objects is relatively small. We say that the entire data space is in the partition level 0, the cells partitioned from it is in the partition level

1, and so on. In Fig. 4, we assume that the page capacity is 4, and if a cell contains at least three points we call it the cluster, otherwise the points are called the outliers. Fig. 4(a) shows the partitions resulted from three times of partitioning in the GC-tree. We have fifteen points. After the first partitioning of the entire space, the points 1, 2, 3, 4, 5, and 6 are identified as outliers and other points are included in a cluster. Thus we store the six points in a single virtual page and we partition again the cluster since the number of points in the cluster exceeds the page capacity. In the second partitioning, points 7, 8, and 9 are identified as outliers, and the points 10, 11, and 12 are identified as outliers in the third partition. In all, we have one cluster page in level 3, three outlier pages in partition levels 2, 1, and 0. Fig. 4(b) shows the result from the traditional partitioning. The query point is denoted by "$\times$" and the query sphere k-NN^{sphere} to find five N\!Ns is depicted by the circle around "$\times$." As depicted, the large search sphere k-NN^{sphere} may intersect most of the partitions if we partition the space in a balanced way since the size of the bounding rectangles are large in high dimensions. However, in the density-based partitioning of the GC-tree, not only the small bounding region of a cluster may avoid being intersected by k-NN^{sphere} but also large number of outliers on the same partition level can be read by a single read of virtual disk page. These are the key performance improvement achieved by the GC-tree. The physical adjancy of the pages chained to form a large virtual page on the disk is implementation dependent and is not discussed in this paper. When going to higher dimensions, the size of bounding rectangles for the traditional partitioning method grows far larger because the possibility to split the data space in each dimension is reduced. On the other hand, for the density-based partitioning, the relative size of bounding rectangles to the whole data space is reduced because we split the space as many as the number of dimensions at a time.

In a dynamic environment, the GC-tree partitions the subspace corresponding to a cluster or a sparse cell at the center of every dimension when the corresponding disk page overflows due to a subsequent insertion. Fig. 5 shows how a 2-D data space is partitioned and thus the corresponding GC-tree grows upon repeated insertion. Initially, there is a single space or cell, which is the whole data space. Conceptually, a number of points are inserted into this cell and there exists a disk page corresponding to this cell. In Fig. 5(a), the data space has already been divided into four cells. Let us assume that the density threshold τ is 3/4, that is, the leaf node fanout is four, a cluster has to contain at least three points and the points lying in the cell that has less than three points are treated as outliers. Fig. 5(a) is the initial state where the database contains four points in the entire data space. There are one root node and one leaf node. The leaf node contains the LPC approximations for four points (outliers) that lie inside the sparse cells. The cell vector of the root node denotes the whole data space. The symbol "–" denotes the entire domain for a dimension in a given cell. Another point is added and a cluster is generated in Fig. 5(b). In Fig. 5(b), a new node B denotes a cluster whose cell vector is (0, 0) and outliers are redistributed in the node A. More points are added and we partition the overflowing space into 2^2 cells and make a new cluster C. New cluster node C is made into a lower level node for the partitioning node. In this example, we assumed that the fanout of the nonleaf node is 2. Fig. 5(c) shows the state of the GC-tree after the partitioning of the space of cell B.

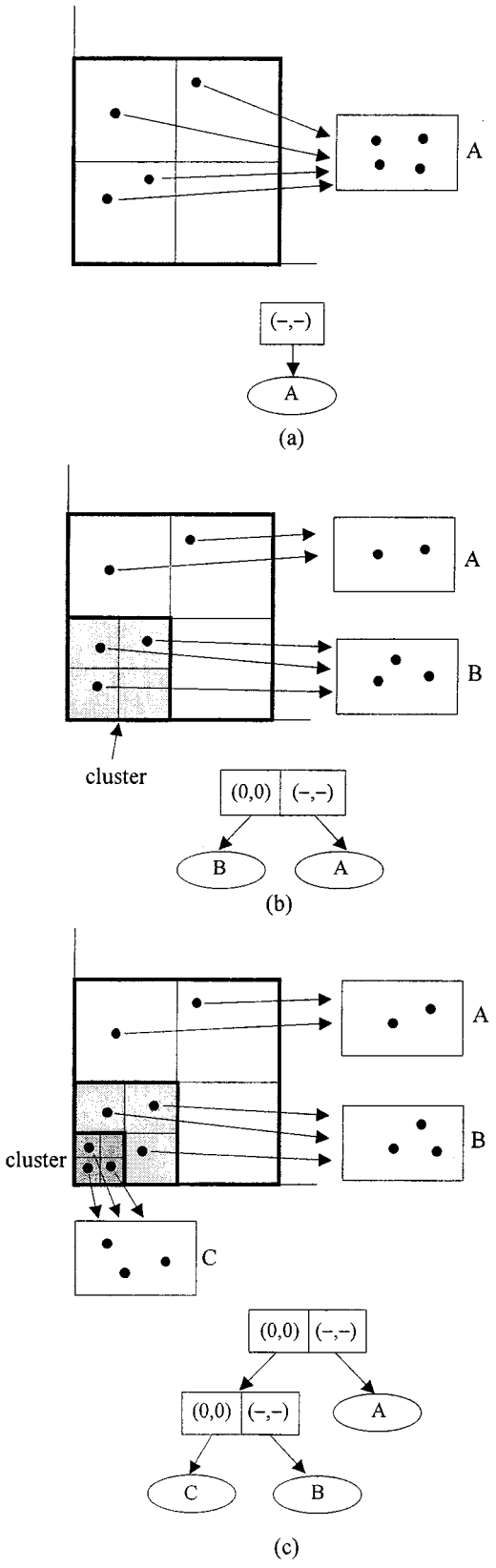

Fig. 5. Space partitioning and dynamic growth of the GC-tree. (a) Initial state. (b) The fifth point is added. (c) After the partiotin of cell B in (b).

number of elements	pointer to parent	cell vector 1	pointer to child 1	...	cell vector i	pointer to child i	pointer to overflow page

Fig. 6. Structure of a nonleaf node.

number of elements	pointer to parent	LPC_1	pointer to object 1	...	LPC_j	pointer to object j	pointer to overflow page

Fig. 7. Structure of a leaf node.

B. Index Structure

The GC-tree is a dynamic index organization that consists of two components: a *directory* and *data nodes*. A directory is composed of nonleaf and leaf nodes. We use the term *directory node* to represent the nonleaf node or the leaf node. The directory nodes are employed for indexing and the data node is used for storing real objects. Each node corresponds to a region (cell) in the data space and is mapped to a disk page. In the GC-tree, there are two types of region corresponding to *clusters* and *outlier regions*. The region corresponding to the cluster is a hyper-rectangle with the sides of equal length. The outlier region is the remaining region after removing the regions for clusters from the original region. However, the cell vector for the outlier region is represented by that of the original region. We use the terms *cluster node* and the *outlier node* for the node corresponding to the cluster and for the node for outliers, respectively.

The entry of the leaf node consists of the LPC approximation $\langle r, \theta \rangle$ of the real object and the pointer to the data node in which the real object is stored. The entry in the nonleaf node contains the *cell vector* corresponding to the region covered by the lower level node and the pointer to the lower level node. Figs. 6 and 7 show the nonleaf and leaf node structures of the GC-tree. LPC_j in the leaf node structure represents the LPCs of an object j.

Example 1: To illustrate the correspondence between the space partition hierarchy and the index tree hierarchy of the GC-tree, consider a 2-D GC-tree with a four-level index in Figs. 8 and 9. In Fig. 8, the 2-D data space has already been partitioned into 4×4 cells by using 2 bits per dimension. The density threshold τ is assumed to be $3/P$, where P is the leaf node fanout. *Root* directory node has three entries. The second directory entry with the cell vector (00, 11) in *Root* points to the node C2 in Fig. 9 and represents the cell C2 in Fig. 8 which, in turn, forms a finer partition into a cluster C4 and an outlier region. Cell C4 in Fig. 8 also forms a finer partition into a cluster C5 and an outlier region, and their corresponding nodes exist in the GC-tree in Fig. 9. Note that data points are approximated by $\langle c_i, r, \theta \rangle$. The LPC information $\langle r, \theta \rangle$ is represented in the leaf node and the cell vector c_i is represented by the index entries of the nonleaf parent of the leaf node. The cell vector in the index entry of the higher node is used as a common prefix of the cell vectors in the lower node. For example, the cell vector (010, 011) of C3 can be obtained by prefixing the cell vector (01, 01) of the higher node index entry to its own cell vector (0, 1). By using this common prefix to represent the lower node cell vector, the GC-tree can increase the fanout of the nonleaf node, and thus reduces the number of disk accesses. Together with the density-based nonuniform partitioning strategy, it is another advantage of the GC-tree for reducing disk I/O cost.

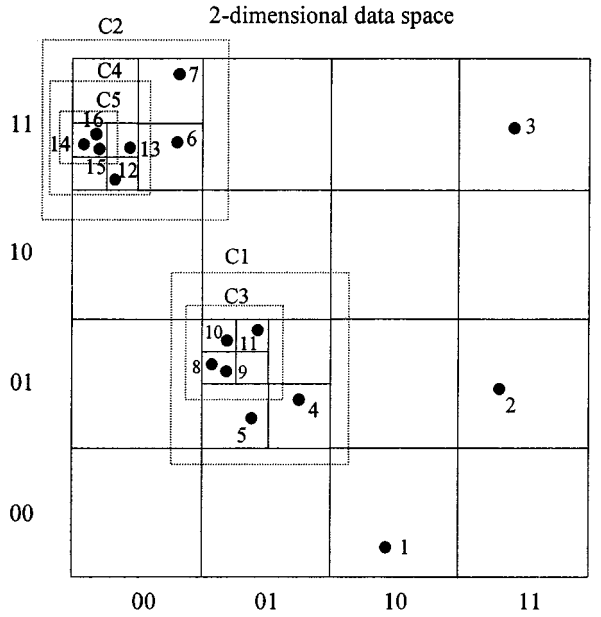

Fig. 8. Nonuniform partition of the data space.

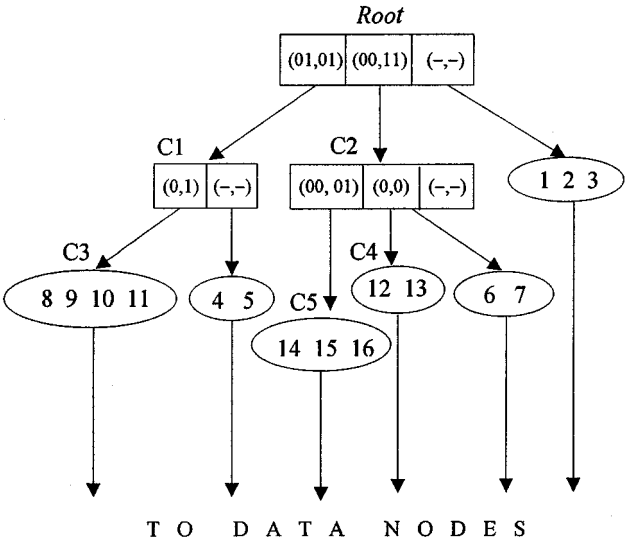

Fig. 9. Index for the partition of Fig. 8.

Algorithm Insert(Node *N*, Object *O*)
// Insert object *O* into the GC-tree rooted at *N*.
{
1. If *N* is NULL {
   // This is the initial state.
   1.1. Create a leaf node with a directory entry (r, θ, *Ptr*), where (r, θ) is the local polar coordinates of the input object *O* and *Ptr* points to the data node of the input object. The local cell vector *Cell* in which the input object lies is stored in the parent nonleaf node and is used to guide the traversal of the GC-tree.
   1.2. Create a nonleaf node with an index entry (*Cell*, *Ptr*), where *Cell* represents the cell vector and each component of it is ì–î, and *Ptr* points to a leaf node containing the local polar coordinates of the object *O*.
   1.3. Let *Root* point to the above nonleaf node.
   }
2. Else {
   2.1. Perform an exact-match query on *Feature Vector* of the input object *O* to choose a leaf node *L* in which to place the input object.
   2.2. If node *L* is not full, Add the object *O* to *L*.
   2.3. Else invoke **SplitLeaf(*L*, *O*)**.
   }
}

Fig. 10. Algorithm Insert.

IV. Algorithms

This section describes the algorithms for building the GC-tree and searching k NNs on the GC-tree.

A. Insertion

We limit our discussion to the insertion of a single object to the level of leaf node. Generalizing this algorithm for the case of more than one object and to the level of data node can be done in an obvious manner. In addition, we omit the deletion algorithm since it is very similar to the insertion algorithm unless deletion causes an underflow. In this case, the remaining branches of the node are deleted and reinserted.

We define some parameters that are used as program variables in describing the algorithms. *Root* specifies the pointer to the root level directory node of the GC-tree. *Feature Vector* and *LPC* are the feature vector and the LPCs of the object to be inserted, respectively. $Cell$ is the cell vector representing the cell and Ptr points to the lower level directory node corresponding to the cell or to the data node.

The algorithm **Insert** in Fig. 10 descends the GC-tree to locate the suitable leaf node for accommodating a new object, possibly causing a split if the leaf is full. Unlike any other dynamic index tree, the GC-tree does not grow in a bottom-up fashion. The overflow of a node N is managed by allocating a new node N' at the same or lower level of N, identifying clusters and outliers among entries, distributing entries to the two nodes or only the input object to the newly allocated node N', and posting directory entries to the parent node if necessary. The node split algorithms can be found in Figs. 11 and 12.

When a node overflows due to the insertion of an outlier point or a nonleaf entry posted (promoted) from the outlier node, the GC-tree allocates a new node N', inserts the entry to N', and simply links N' to the overflowing node. It makes multiple nodes a single virtual node. This is based on the observation

Algorithm SplitLeaf(Node *N*, Object *O*)
{
1. Allocate a new node N.
2. Partition the subspace covered by the overflowing node N based on the space partition strategy of the GC-tree, and identify the cluster and the outliers.
3. If a cluster is found {
   3.1. Move the outliers to the node N and move the cluster points to node N.
   3.2. Construct directory entries E and E for nodes N and N, respectively.
   3.3. Let N_p be the parent node of the node N.
   3.4. If N_p is not full, store entries E and E in N_p.
   3.5. Else Invoke **SplitNonleaf(N_p, E, E)**.
   }
4. Else {
   4.1. Insert the object *O* into node N.
   4.2. Chain the node N to node N.
   }
}

Fig. 11. Algorithm SplitLeaf.

Algorithm SplitNonleaf(Node *N*, Entry *E*, Entry *E*)
{
1. Allocate a new node N.
2. Let E_p be the directory entry in N that points to the overflowed node.
3. If E_p points to the cluster node {
   3.1. Add E and E to N.
   3.2. Adjust E_p so that it points to N.
   3.2. Write nodes N and N.
   }
4. Else { // E_p points to the outlier node. //
   4.1. Replace E_p with E.
   4.2. Insert E to N.
   4.3. Chain the node N to node N.
   4.4. Write nodes N and N.
   }
}

Fig. 12. Algorithm SplitNonleaf.

that: 1) the volume covered by the outliers is so large that it may not be pruned in the search, and thus the GC-tree reads all outliers on the same cell at a time; and 2) the number of clusters on a cell is may be large that all nonleaf entries pointing them cannot not be accommodated in a nonleaf node, but the GC-tree needs to read them all at a time because we have to examine all candidates under a node.

If a cluster node N overflows due to the insertion of a cluster point (i.e., the input point that generates a cluster after the node partition), the GC-tree allocates a new node N', identifies cluster points and outliers, distributes outliers to N and cluster points to N', and promotes the directory entries for N and N' to the parent node.

B. k-NN Search

The k-NN search algorithm in Figs. 14 retrieves k NNs of a query vector $\boldsymbol{q}$. It consists of two stages as in the VA approach. However, unlike the VA approach that linearly scans the whole approximation file, the GC-tree can use a branch-and-bound technique similar to the one designed for the R-tree [21]. The GC-tree utilizes three global structures: two priority queues

bl_list and *cand_list*, and a k-element array knn. The knn maintains the k NNs processed to that point and, at the end of execution, contains the final result. The *bl_list* contains branches of nonleaf nodes with the minimum and maximum distances to those from the query point. The *cand_list* maintains a set of candidates, i.e., qualifying objects. The *bl_list* and the *cand_list* are implemented with a *min heap* [11]. Also in the *cand_list*, the lower bound $d_{\min}$ and the upper bound $d_{\max}$ on the distance of each candidate to the query vector are kept. Since the real distance between the query vector and a point represented by approximation cannot be smaller than the minimum distance between the query vector and the approximation, the real distance is lower-bounded by $d_{\min}$. Similarly, the real distance is upper-bounded by $d_{\max}$ since it cannot be larger than the maximum distance between the query vector and the approximation. If an approximation is encountered such that its $d_{\min}$ exceeds the k-th smallest upper bound found so far, the corresponding vector can be eliminated safely since k better candidates already exist.

The routing information used in the nonleaf node and the leaf node of the GC-tree is different: the cell vector and LPC are used for the nonleaf node and the leaf node, respectively. Therefore, the distance bounds $d_{\min}$ and $d_{\max}$ are also different between them. The bounds $d_{\min}$ and $d_{\max}$ for the nonleaf node are determined as follows:

$$d_{\min}^2 = \sum_{i=1}^{d} l_i^2 \quad \text{where } l_i = \begin{cases} q_i - m[r_{pi}+1] & r_{pi} < r_{qi} \\ 0 & r_{pi} = r_{qi} \\ m[r_{pi}+1] - q_i & r_{pi} > r_{qi} \end{cases}$$

$$d_{\max}^2 = \sum_{i=1}^{d} u_i^2$$

$$\text{where } u_i = \begin{cases} q_i - m[r_{pi}] & r_{pi} < r_{qi} \\ \max(q_i - m[r_{pi}],\ m[r_{pi}+1] - q_i & r_{pi} = r_{qi} \\ m[r_{pi}+1] - q_i & r_{pi} > r_{qi} \end{cases}$$

where q_i is the component of $\boldsymbol{q}$ in i-th dimension and lies within the cell r_{qi}, r_{pi} is a cell into which the database point $\boldsymbol{p}$ falls in dimension i, and $m[j]$ is the j-th mark in a given dimension, i.e., the starting position of the j-th partition in the dimension.

For the leaf node, $d_{\min}$ and $d_{\max}$ are computed as follows:

$$d_{\min}^2 = |\boldsymbol{p}|^2 + |\boldsymbol{q}|^2 - 2|\boldsymbol{p}||\boldsymbol{q}|\cos|\theta_1 - \theta_2|$$
$$d_{\max}^2 = |\boldsymbol{p}|^2 + |\boldsymbol{q}|^2 - 2|\boldsymbol{p}||\boldsymbol{q}|\cos(\theta_1 + \theta_2)$$

where $\theta_1 (= \angle AOD)$ is the angle between the vector $\boldsymbol{p}$ and the diagonal of the cell in which $\boldsymbol{p}$ lies and $\theta_2 (= \angle BOD)$ is the angle between the vector $\boldsymbol{q}$ and the diagonal of the cell (as illustrated in Fig. 13). In other words, $d_{\min}$ and $d_{\max}$ are determined when the angle $\phi (0° \le \phi \le 180°)$ between two vectors $\boldsymbol{p}$ and $\boldsymbol{q}$ is minimum and maximum, respectively. The minimum angle and the maximum angle between two vectors $\boldsymbol{p}$ and $\boldsymbol{q}$ are determined by $|\theta_1 - \theta_2|$ and $(\theta_1 + \theta_2)$, respectively

In the first search stage, the algorithm **k_NN_Search** in Fig. 15 examines the top-level branches of the GC-tree, computes $d_{\min}$ and $d_{\max}$ for each branch, and traverses the most promosing branch with the depth first order. At each stage of traversal, the order of search is determined by the increasing

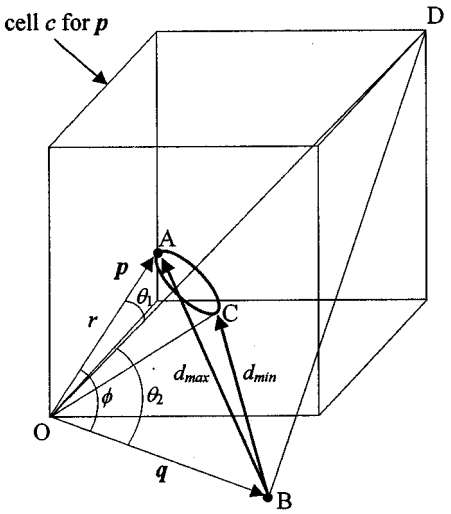

Fig. 13. Lower bound $d_{\min}$ and upper bound $d_{\max}$ for a leaf node in a three-dimensional data space.

Algorithm k_NN_Query(Query *q*, Integer *k*)
{
// *MAX*: a value that exceeds the possible largest distance between any two points in a database //
// NN: nearest neighbor //
// *k*-NN: *k*-th nearest neighbor //
// *oid*(*x*): object identifier of *x* //
// *dist*(*x*): distance between *x* and ***q*** //
// d_{min}(*x*): d_{min} between *x* and ***q*** //
// *bl_list*: min heap to maintain the branch list //
// *cand_list*: min heap to maintain the candidate set //
// *bl*: a variable that represents an element in *bl_list*; it consists of object no, d_{min} and d_{max}. //
// *knn*: *k*-element array with the element [*oid*(*x*), *dist*(*x*)] //
// The insertion into *knn* is an ordered insertion, i.e., the new element *x* is inserted into the correct position in *knn* with respect to *dist*(*x*). //

1. `for` *i* := 0 `to` *k* `do` {
   knn[*i*].*dist* := *MAX*.
   }
2. *k-NN*^{dist} := *MAX*.
3. *bl.node_no* := *Root*;
   // *Root* is the number of root node of the GC-tree

// stage 1
4. `do` {
   next_node := **ReadNodeFromGCtree**(*bl.node_no*);
   k_NN_Search(*next_node*, *q*, *k*);
   } `while` (**GetFromBranchList**(&*bl*) ≠ NULL);

// stage 2
5. `for` *i* := 0 `to` *k* `do` {
   knn[*i*].*dist* := *MAX*;
   }
6. `while` (get the candidate *O* from *cand_list* and $d_{min}(O) \le$ *k-NN*^{dist}) `do` {
   6.1. Read original vector ***p*** corresponding to *O*.
   6.2. If (*dist*(***p***) < *k-NN*^{dist}) {
   6.2.1. Insert the NN [*oid*(*O*), *dist*(***p***)] into *knn*.
   6.2.2. *k-NN*^{dist} := *dist*(*k*-NN in *knn*).
   }
   }
}

Fig. 14. Algorithm **k_NN_Query**.

```
Algorithm k_NN_Search(Node N, Query q, Integer k)
{
1. If N is a leaf node {
     For every entry N_i in N {
       1.1. Compute d_min and d_max between N_i and the query
            object q.
       1.2. If (d_min ≤ k-NN^dist) {
              1.2.1. Insert the [oid(N_i), d_min, d_max] into cand_list.
              1.2.2. If (d_max < k-NN^dist) {
                       Insert the NN [oid(N_i), d_max] into knn.
                       k-NN^dist := dist(k-NN in knn).
                     }
            }
     }
   }
2. Else {
     For every entry N_i in N {
     2.1. Compute d_min and d_max between N_i and the query
          object q.
     2.2. Insert [N_i, d_min] into the min-heap bl_list.
     2.3. If (d_max < k-NN^dist) {
            Insert the NN [ptr(N_i), d_max] into knn.
            k-NN^dist := dist(k-NN in knn).
          }
     }
   }
}
```

Fig. 15. Algorithm **k_NN_Search**.

order of $d_{\min}$. Since the pruning criterion of k-NN search is dynamic – the search radius is the distance k-NN^{dist} between $\boldsymbol{q}$ and its current k-th NN – the order in which nodes are visited can affect the search performance. If objects are found whose $d_{\min}$ exceeds k-NN^{dist}, then we can safely eliminate them since k better candidates have already been found. At the end of the first stage, the *cand_list* contains a set of candidates.

In the second stage, as in the VA approach, it refines the candidate set by visiting real vectors themselves in increasing order of $d_{\min}$. In this stage, not all remaining candidates are visited. Rather, this stage ends when an approximation is encountered whose lower bound exceeds or equals the k-th distance k-NN^{dist} in the answer set, and the final k nearest vectors in the answer set are the search result.

The function **ReadNodeFromGCtree** reads a disk page corresponding to the node number provided by the input argument from the GC-tree. The function **GetFromBranchList** fetches a branch list element with the smallest $d_{\min}$ entry from the *bl_list* to the variable *bl*.

C. Cost Estimation of the Algorithms

Now let us consider the costs for the insertion of an object and for the k-NN search in the GC-tree. The parameters for cost analysis are defined as follows: n is the total number of nodes in the GC-tree, f is the average fanout of the nonleaf node, and the average length of the chain of each node is m, that is, m nodes are linked in a single virtual node. For the nonleaf node, the larger the number of clusters, the larger the value of m, and in this case, the larger m is favorable to the system performance because it may increase the opportunity of pruning nodes in the search. For the outlier node among leaf nodes, the larger m means that there are lots of outliers, and this has a negative effect on the system performance. For the uniformly distributed dataset, in fact, all leaf nodes are linked in a single virtual node. In other words, the GC-tree degenerates to the LPC-file.

The insertion cost consists of the cost to find a leaf node into which to insert the input object and the cost to update the nodes affected by the insertion. The cost to find the leaf node is $O((1+\lfloor \log_f n \rfloor)\cdot m)$. The update cost to reflect the insertion is only one disk page write if there is no page overflow. If the page overflow occurs, the space partition to identify clusters and outliers is required and the parent node is also updated. The CPU time to perform the space partition is $O(P \cdot l)$ where P is the leaf node fanout and l is the maximum partition level. l corresponds to the maximum number of bits to represent a subspace cell and is 16 in our implementation. In other words, the GC-tree identifies clusters and outliers by iterating at worst l times for each object in the overflowing node. The number of maximum page writes is three (the writes of the overflowed node, the newly allocated node, and the parent node). Therefore, the total cost for a single object insertion is the I/O cost of $O((1+\lfloor \log_f n \rfloor)\cdot m+3)$ disk accesses plus the CPU cost of $O(P \cdot l)$.

The k-NN search algorithm of the GC-tree is the combination of the branch-and-bound technique developed for the R-trees and the method for the VA approach. The k-NN search cost in the GC-tree depends mainly on how many nodes are visited (or pruned) in the first stage of the algorithm and how many real objects are visited in the second stage of the algorithm. The number of nodes of the GC-tree visited during the k-NN search can be estimated using the equation given by [16].

To find k NNs the algorithm **k_NN_Query** repeats, in the first stage, the process of computing the distances to lower nodes from current node and storing the branches to branch list, the candidates to the candidate set, and k NNs found so far to the array knn. Let the total number of nodes in the GC-tree be n, the average number of branches in the branch list be l, and the average number of candidates in the candidate set be m. The size of the knn array is k. Thus the complexity of the first stage of the algorithm is $O(n \cdot \max(\log m, \log l)) + O(k)$ since the time complexity of the insertion and deletion for mean-heap is $O(\log s)$, where s is the number of elements in the min-heap, and the cost to insert an object to knn is $O(k)$. The cost for the second stage is $O(m \cdot (\log m + k))$.

V. Performance Evaluation

To demonstrate the practical effectiveness of the new indexing method, we performed an extensive experimental evaluation of the GC-tree and compared it with the competitors: the IQ-tree, the VA-file, the LPC-file, and the linear scan. Our experiments have been computed under the Microsoft Windows 2000 on Intel Pentium III 800 MHz processor with 256 MB of main memory.

For our experiments, we used 13 724 256-color images of U.S. stamps and photos in IBM QBIC image database. Stamps often come in series (e.g., states, birds, flowers) with common colors and related designs, and the U.S. post office has often used similar colors for many long-running stamps. As a result, this real image dataset shows a *highly clustered distribution*.

For a number of the experiments we performed, datasets containing far more than 13 724 image vectors were required. To obtain larger databases, the 13 724 256-dimensional data were synthetically scaled up to 100 000 vectors, while retaining the original distribution of the image dataset. To generate a new image vector v, we randomly choose a vector u from the original dataset and find a cell c, which is either dense or sparse, in which u lies. We then select two out of the vectors in the cell c and average them for each dimension i, $0 \leq i \leq 255$. The averaged vector is stored as a new vector v.

We also performed the experiments on the 256-dimensional random dataset which follows the random distribution and the 256-dimensional skewed dataset which follows the skewed distribution according to Zipf's law [20]. The Zipf distribution is defined as follows, and the value of z used is 0.5:

$$f(i) = \frac{\frac{1}{i^z}}{\sum_{j=1}^{N} \frac{1}{j^z}}, i = 1, 2, \ldots, N.$$

In all experiments, the Euclidean distance metric L_2 was used, and the number of nearest neighbors to find was always ten, i.e., $k = 10$. The page size used in the experiment was 8 KB. One-thousand 10-NN queries were processed and the results were averaged. The query vector $\boldsymbol{q}$ was randomly selected from the scaled-up image dataset. The density threshold τ used was 8/15. In other words, the fanout of a leaf node is 15 and we regard a partitioned cell as a cluster if it contains at least eight points.

A. Pruning Rate of Directory Nodes

Most tree-structured multidimensional indexing methods fail to prune directory nodes during the k-NN search due to the large volume of k-NN^{sphere} and large bounding regions caused by the inherent sparsity of the high-dimensional space. Therefore, the pruning rate of directory nodes during the k-NN search can be a good indicator to estimate the performance of the index structure. Fig. 16 shows the result of the experiments for the directory node pruning rate. In the real image dataset, the directory node pruning rate of the GC-tree is more than 60%. From this result the GC-tree can be expected to provide a good search performance in the highly-clustered dataset such as real images. However, the experimental results in the Zipf and randomly distributed datasets shows that the pruning rate is not as good as in the real datasets. For the random dataset, we could not prune any node during the search. In all datasets, the GC-tree outperforms the IQ-tree. The good pruning rate of the GC-tree comes from the tight bounding regions that accommodate only the clustered points. In this experiment, the LPC-file and the VA-file were not included since they read the whole approximation file.

B. Vector Selectivity

In the k-NN search algorithm, the vector selectivity for the first stage is the ratio of vector approximations remained without being pruned after the first stage. The real vectors corresponding to the remained approximations have the potential of being accessed in the second stage. The vector selectivity for the second stage is the ratio of vectors accessed during the second stage to the total number of vectors. The vector selectivity is also a good estimator to predict the search performance because it can estimate the number of disk accesses for the real vectors that must be read.

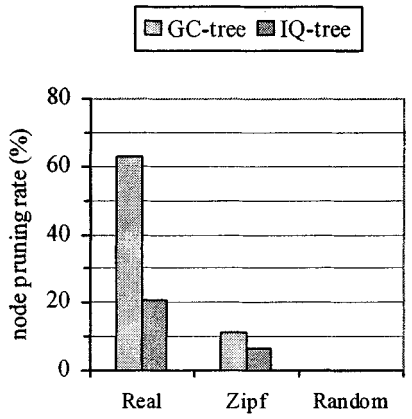

Fig. 16. Pruning rate of directory nodes.

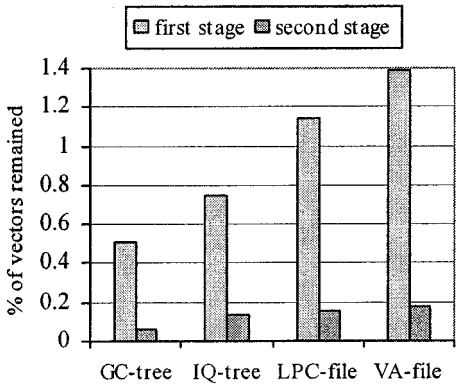

Fig. 17. Vector selectivity experiments (real image dataset).

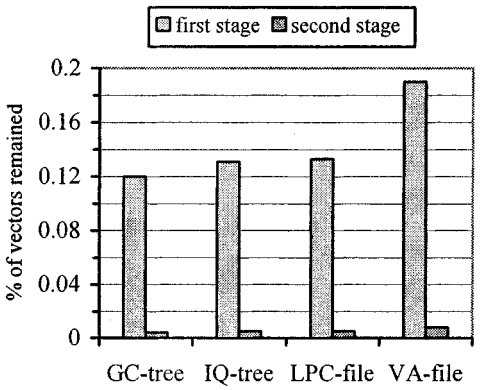

Fig. 18. Vector selectivity experiments (Zipf dataset).

Figs. 17–19 show the results of the vector selectivity experiments in real image dataset, Zipf dataset, and random dataset, respectively. The vertical axis denotes the vector selectivity in the first and the second stages. As shown, the vector selectivity of the GC-tree is better than those of the IQ-tree, the LPC-file, and the VA-file. This superiority comes from the fact that the GC-tree adaptively partitions the data space to find subspaces with high-density clusters and to assign relatively more bits to them. Therefore, the discriminatory power of the approximation

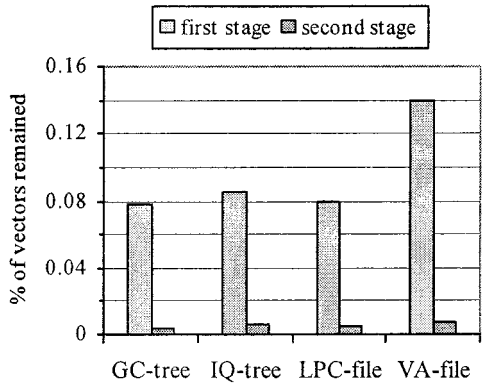

Fig. 19. Vector selectivity experiments (random dataset).

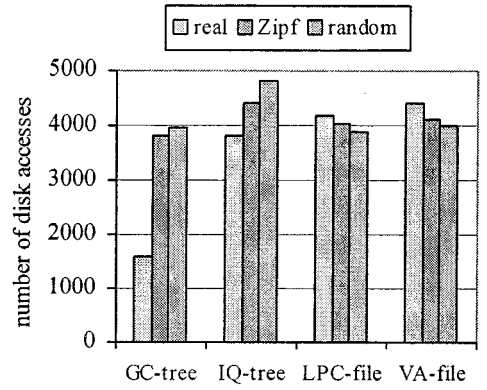

Fig. 20. Disk access experiments.

of the GC-tree increases. This means that the GC-tree preserves the advantage of the VA approach that accesses only a small fraction of the real vectors. Although the focus of the GC-tree is to apply the advantage of the hierarchical indexing technique to the VA approach, it also improves the vector selectivity since it adaptively assigns the number of bits to the cell vector (i.e., the vector approximation).

C. Number of Disk Accesses

While the directory node pruning rate and the selectivity experimental results can be used to estimate the search performance, it does not fully reflect the search performance because it lacks the real cost of reading the approximation file or the index file. We computed the total number of disk accesses for the k-NN search that include the accesses for real vectors, approximations, and an index (in the case of the GC-tree and the IQ-tree). Fig. 20 shows the total number of disk accesses actually performed to find ten NNs. For the real image dataset, the number of disk accesses performed by the GC-tree is far smaller that those of the IQ-tree, the LPC-file and the VA-file, respectively. The performance improvement of the GC-tree comes from both of the density-based space partitioning and the hierarchical index structure supporting the partitioning strategy. This result shows that the hierarchical index structure can be successfully employed for indexing high-dimensional data by applying elaborate partitioning strategy based on data analysis. For the Zipf and random datasets, the performance of the GC-tree is somewhat better and similar to the vector approximation techniques (LPC-file and the VA-file) because there is not much chance to prune the directory nodes.

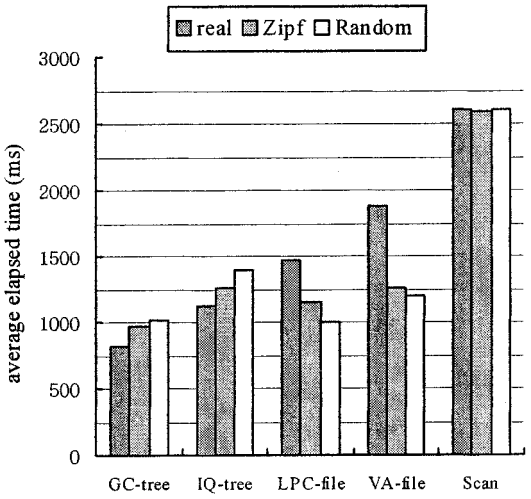

Fig. 21. Comparison of the average elapsed time for 10-NN search.

D. Elapsed Time Experiments

To demonstrate the practical effectiveness of the GC-tree, we performed a number of timing experiments. Fig. 21 shows the elapsed time of the 10-NN search. The *Scan* in the horizontal axis is the linear scan that sequentially scans the real vectors themselves, maintaining a ranked list of the k NN vectors encountered so far. While the I/O patterns generated by the k-NN search algorithm are inherently random, the linear scan can save much disk startup time to begin the read. A well-tuned linear scan algorithm frequently outperforms more sophisticated indexing methods which frequently generate the random disk access, and thus the linear scan can be generally used as the yardstick for performance comparison in high dimensions.

Fig. 21 shows that the GC-tree achieves a remarkable speed-up over the IQ-tree, the VA techniques, and the linear scan. Summarizing the results of our all experiments, we make the observation that the GC-tree outperforms the competitors.

VI. Conclusion

In this paper, we proposed a new dynamic index structure called the GC-tree for efficient similarity search in high-dimensional image databases. It is based on the density-based space partitioning and the index structure that reflects the partition hierarchy. The performance evaluation demonstrated the effectiveness of our technique by comparing the state-of-the-art VA techniques and the IQ-tree. The design goal of the GC-tree is to combine the advantages of the VA approach and the multidimensional index structure. However, two approaches are not readily combined because they have different design principles and goals. Until now, the multidimensional index structures based on the conventional data partitioning have been defeated by the high dimensionality. However, the GC-tree achieves the performance improvement over both approaches by combining them based on the careful data analysis.

As a future study, we are considering the technique to control the density threshold adaptively based on the data distribution.

REFERENCES

[1] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, "An optimal algorithm for approximate nearest neighbor searching in fixed dimensions," *J. ACM*, vol. 45, no. 6, pp. 891–923, Nov. 1998.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles," in *Proc. ACM SIGMOD ICMD*, 1990, pp. 322–331.

[3] S. Berchtold, C. Boehm, H. V. Jagadish, H.-P. Kriegel, and J. Sander, "Independent quantization: An index compression technique for high-dimensional data spaces," *Proc. IEEE Data Engineering*, pp. 577–588, 2000.

[4] S. Berchtold, B. Ertl, D. A. Keim, H.-P. Kriegel, and T. Seidl, "Fast nearest neighbor search in high-dimensional space," *Proc. IEEE Data Engineering*, pp. 209–218, 1998.

[5] S. Berchtold, D. A. Keim, and H.-P. Kriegel, "The X-tree: An Index structure for high-dimensional data," in *Proc. ICVLDB*, 1996, pp. 28–39.

[6] G.-H. Cha, X. Zhu, D. Petkovic, and C.-W. Chung, "An efficient indexing method for nearest neighbor searches in high-dimensional image databases," *IEEE Trans. Multimedia*, vol. 4, pp. 76–87, Mar. 2002.

[7] G.-H. Cha and C.-W. Chung, "A new indexing scheme for content-based image retrieval," *Multimedia Tools Applicat.*, vol. 6, pp. 263–288, May 1998.

[8] K. Chakrabarti and S. Mehrotra, "Local dimensionality reduction: A new approach to indexing high-dimensional spaces," in *Proc. ICVLDB*, 2000, pp. 89–100.

[9] M. Flickner, H. Sawhney, W. Niblack, 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 Comput.*, vol. 28, pp. 23–32, Jan. 1995.

[10] A. Hinneburg, C. C. Aggarwal, and D. A. Keim, "What is the nearest neighbor in high-dimensional spaces?," in *Proc. ICVLDB*, 2000, pp. 506–515.

[11] E. Horowitz, S. Sahni, and D. Mehta, *Fundamentals of Data Structures in C++*. Rockville, MD: Computer Science, 1995.

[12] P. Indyk and R. Motwani, "Approximate nearest neighbors: toward removing the curse of dimensionality," in *Proc. ACM STC*, 1998, pp. 604–613.

[13] K. V. R. Kanth, D. Agrawal, and A. Singh, "Dimensionality reduction for similarity searching in dynamic databases," in *Proc. ACM SIGMOD ICMD*, 1998, pp. 166–176.

[14] N. Katayama and R. Satoh, "The SR-tree: An index structure for high-dimensional nearest neighbor queries," in *Proc. ACM SIGMOD ICMD*, 1997, pp. 369–380.

[15] E. Kushilevitz, R. Ostrovsky, and Y. Rabani, "Efficient search for approximate nearest neighbor in high-dimensional spaces," in *Proc. ACM STC*, 1998, pp. 614–623.

[16] J.-H. Lee, G.-H. Cha, and C.-W. Chung, "A model for k-nearest neighbor query processing cost in multidimensional data space," *Inf. Process. Lett.*, vol. 69, pp. 69–76, 1999.

[17] K.-I. Lin, H. V. Jagadish, and C. Faloutsos, "The TV-tree: An index structure for high-dimensional data," *VLDB J.*, vol. 3, no. 4, pp. 517–542, 1994.

[18] N. Megiddo and U. Shaft, "Efficient nearest neighbor indexing based on a collection of space-filling curves," IBM Almaden Research Center, San Jose, CA, RJ 10 093, 1997.

[19] M. Miyahara and Y. Yoshida, "Mathematical transform of (R,G,B) color data to Munsell (H, V, C) color data," *Vis. Commun. Image Process.*, vol. 1001, pp. 650–657, 1992.

[20] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, P. Yanker, C. Faloutsos, and G. Taubin, "The QBIC project: Querying images by content using color, texture, and shape," in *Proc. SPIE Conf. Storage and Retrieval for Image and Video Databases II*, 1993, pp. 173–187.

[21] N. Roussopoulos, S. Kelly, and F. Vincent, "Nearest neighbor queries," in *Proc. ACM SIGMOD ICMD*, 1995, pp. 71–79.

[22] J. Shepherd, X. Zhu, and N. Megiddo, "A fast indexing method for multidimensional nearest neighbor search," in *Proc. IS&T/SPIE Conf. Storage and Retrieval for Image and Video Databases VII*, 1999, pp. 350–355.

[23] R. Weber, H.-J. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," in *Proc. ICVLDB*, 1998, pp. 194–205.

Guang-Ho Cha received the Ph.D. degree in computer engineering from the Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 1997.

From 1999 to 2000, he was a Visiting Scientist at the IBM Almaden Research Center, San Jose, CA. He was also an Assistant Professor of Multimedia Engineering, Tongmyong University of Information Technology, Busan, South Korea. He is currently an Assistant Professor in the Department of Multimedia Science at the Sookmyung Women's University, Seoul, South Korea. His research interests include content-based image/video/music indexing and retrieval, XML databases, and distance learning.

Chin-Wan Chung received the Ph.D. degree from the University of Michigan, Ann Arbor, in 1983.

He was a Senior Research Scientist and a Staff Research Scientist in the Computer Science Department at the General Motors Research Laboratories (GMR). While at GMR, he developed DATAPLEX, a heterogeneous distributed database management system integrating relational databases and hierarchical databases. Since 1993, he has been a Professor in the Division of Computer Sciences at the Korea Advanced Institute of Science and Technology (KAIST), South Korea. At KAIST, he developed a full-scale object-oriented spatial database management system call OMEGA, which supports ODMG standards. His current research interests include XML, multimedia databases, spatio-temporal databases, and Web databases.