Content-Based Indexing and Retrieval in Large Image Databases

Guang-Ho Cha and Chin-Wan Chung

Abstract

In this paper, we propose a new access method, called the HG-tree, to support indexing and retrieval by image content in large image databases. Image content is represented by a point in a multidimensional feature space. The types of queries considered are the range query and the nearest-neighbor query, both in a multidimensional space. Our goals are twofold: increasing the storage utilization and decreasing the area covered by the directory regions of the index tree. The high storage utilization and the small directory area reduce the number of nodes that have to be touched during the query processing. The first goal is achieved by absorbing splitting if possible, and when splitting is necessary, converting two nodes to three. The second goal is achieved by maintaining the area occupied by the directory region minimally on the directory nodes. We note that there is a trade-off between the two design goals, but the HG-tree is so flexible that it can control the trade-off. We present the design of our access method and associated algorithms. In addition, we report the results of a series of tests, comparing the proposed access method with the buddy-tree, which is one of the most successful point access methods for a multidimensional space. The results show the superiority of our method.

I. Introduction

Image databases are becoming increasingly popular with many applications such as medical databases, trademark and copyright databases, CAD/CAM, geographic information systems, and digital libraries. One of the key issues of these areas is content-based retrieval which helps users to retrieve relevant images based on their contents. To provide this feature effectively, it is essential to develop an efficient access method that provides fast retrieval.

Content-based retrieval can be achieved with two approaches. The first is to create a set of attributes, keywords, or text annotations manually that describe the content of the image, and then queries are specified using those properties. The second makes use of the visual content of the image. It extracts a set of visual features of the images, and then queries are processed by searching those features. Some examples of such features are color, texture, shape and so on. In the first approach, conventional access methods such as the B-tree [3] and the signature files [8] can be used. Retrieval by visual features requires more sophisticated access method because the features lie in the domain space with increased dimensionality, i.e., n dimensions. We will focus our attention on the access methods based on the visual features.

An image is characterized by a feature vector f comprising of n features, i.e., $f = (f_1, f_2, ..., f_n)$, such that f_i belongs to D_i, $i = 1, ..., n$. Sets D_i, $i=1, ..., n$, are the *domains* of the feature vector, from which a value for the feature can be drawn. The *domain space* is defined as a Cartesian product, $D_1 \times D_2 \times ... \times D_n$, of the domains of all organizing features. Thus an image with n features is represented by a point in an n-dimensional domain space. We call any subset of the domain space a *region*. Generally speaking, the images, or points within the same region have some features in common and the regions are in one-to-one correspondence with the index entries. In this paper we use the terms domain space and feature space interchangeably. We can formulate the content-based image indexing problem as a multidimensional point indexing problem. We distinguish basically between point access methods (PAMs), such as K-D-B tree [23] and buddy-tree [26], and spatial access methods (SAMs), such as R-tree [11] and R*-tree [1], which are designed to handle multidimensional point data and spatial data, respectively. We will focus our attention on multidimensional PAM, because each image object intrinsically corresponds to a point in a multidimensional feature space.

Manuscript received February 3, 1996; accepted May 30, 1996.
The authors are with Department of Information and Communication Engineering, Korea Advanced Institute of Science and Technology, Seoul, Korea.

Unlike traditional databases, which normally retrieve objects through the specification of exact queries which are based on the notion of equality, the types of queries typical in the image databases are based on the notion of the similarity [17]. For example, Find all images that are similar to a given image, within a user-specified tolerance (a range query), or Given an image, find the 5 most similar images (a nearest neighbor query). Similarity queries correspond to range queries or nearest-neighbor queries. Partial match queries are treated as a special case of general range queries. Thus we formulate the content-based image retrieval problem as a nearest neighbor or range search problem.

In this paper, we present a new multidimensional PAM, named the HG-tree, which has taken into account the content-based image retrieval and we show how to search for nearest neighbors as well as for ranges. The paper is organized as follows. Section 2 surveys related work. Section 3 presents the ideas and properties of the HG-tree and its associated algorithms. Section 4 gives the experimental results and analysis which show the superiority of the HG-tree to the buddy-tree. Section 5 makes the conclusions.

II. Related Work

The basic principle of multidimensional PAM is to divide the *n*-dimensional feature space into several regions, each containing not more than a fixed number of entries. Each region corresponds to one disk page and, upon becoming full, is split into two. Since all of the multidimensional access methods are characterized by the way they divide the domain space and the way they represent the divided regions, we classify the multidimensional PAMs according to the following two properties: whether the domain space is divided into rectangles or not, and whether the division into regions is complete or not, i.e. the union of all regions spans the complete data space or not. According to this classification we can classify all known PAMs including the HG-tree into four classes (see Table 1).

Table 1. Classification of multidimensional PAMs.

Class	property		Point Access Methods
	rectangular	complete	
C1	√	√	grid file [18], BMEH-tree [20], K-D-B-tree [23], MB+-tree [28]
C2		√	BANG file [9], hB-tree [16], BV-tree [10], zkdb tree [19]
C3	√		buddy-tree [26], multilevel grid file [27], G-tree [15]
C4			HG-tree

The PAMs in class C1 perform rather efficiently for uniform and uncorrelated data. However, for highly correlated data their performance degenerates. For example, the directory of the grid file grows exponentially with the dimensionality if a strong correlation among attributes exists and the K-D-B-tree suffers the *cascade splitting* problem, that is, the split of one index node causes descendent nodes to be split as well. The consequences of a single insertion are thus wholly unpredictable.

The PAMs in class C2 adapt to the clustering of objects in the data space by allowing more general shapes of directory region, i.e., not rectangles. However, they require the representation of the whole data space, that is, the union of all divided regions spans the complete data space. This suffers a perfor-mance loss in range queries because the directory area overlapping the query region increases.

The approach adopted by the PAMs in class C3 is to maintain the directory region compactly. This is good especially in the distributions where large portions of empty data space occur. However, since its region split policy is severely restricted, it cannot balance the occupancy of the nodes.

The HG-tree in class C4 attempts to solve above problems by allowing more general shape of regions like the PAMs in the class C2 and by representing the regions compactly like the PAMs in the class C3. In the following section, we describe the HG-tree and discuss algorithms for insertion, deletion, and searches for nearest neighbors and ranges.

III. HG-Tree

We now discuss the essential properties of the HG-tree. First of all, let us consider the performance factors that determine the query performance. From the analyses of Faloutsos and Kamel [6] and Pagel et al. [21], that can be applied to most of the multidimensional access methods including the HG-tree, the number of nodes D accessed by the range query $\vec{q}$ can be estimated as follows:

$$D(\vec{q}) = \sum_{n=1}^{N} \prod_{i=1}^{d} (x_{i,n} + q_i) \quad (1)$$

where $x_{i,n}$ is the length of the directory region of the node n in the i-th dimension, q_i is the length of the query region in the i-th dimension, d is dimensionality, N is the number of entire nodes, and the domain space is assumed to be normalized to the unit square. The equation (1) says that the efficiency of the index structure is determined by the two parameters N and $x_{i,n}$. In the index tree, since the number of nodes, N and the length of a directory region in one dimensiona, $x_{i,n}$ are determined by the storage utilization and the area of the directory region, respectively, we can conclude that it is the storage utilization and the node coverage of the index structure that determine the

performance of the index tree.

We define the storage utilization and the directory coverage as follows.

Definition 1. *The storage utilization U of a tree T is:*

$$U=\frac{1}{n}\sum_{i=1}^{n}\frac{F_i}{P_i}$$

where F_i is the number of entries in node i, the P_i is the maximum number of entries that a node i can have, and n is the number of nodes in the tree.

Definition 2. *The node coverage C_n of node i and the directory coverage C_d of tree T are:*

$C_n(i)$ = the area spanned by all the entries enclosed in the node i,

$C_d(T) = \bigcup_{i=1}^{k} C_n(i)$, where k is the number of nodes in the tree T

The major aims of the HG-tree are to increase the storage utilization and to decrease the directory coverage. To achieve the first goal we use a space filling curve, and specifically, the Hilbert curve to apply a linear ordering on the data objects and on the directory regions. A *space-filling curve* is a mapping that maps the unit interval onto the n-dimensional unit hypercube continuously. The path of space-filling curve provides a linear ordering on the grid points retaining some of the spatially associative properties of the space. The Peano curve (also known as the Z curve) [22], the Hilbert curve [12], and the Gray-code curve [5] are examples of space filling curves. Faloutsos and Roseman [7] and Jagadish [13] showed that the Hilbert curve achieves the better clustering than the others. The basic Hilbert curve on a 2×2 grid, denoted by H_1, and the Hilbert curve of order 2, denoted by H_2, are shown in Fig.1. The Hilbert curve can be generalized for higher dimensionalities.

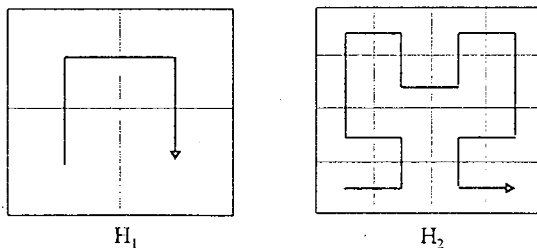
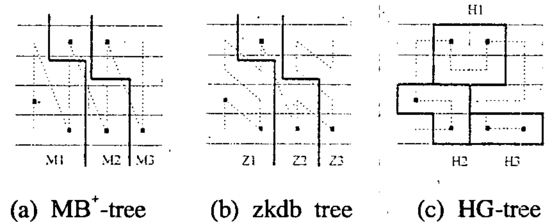


Fig. 1. Hilbert Curves of order 1 and 2.

The zkdb tree, G-tree, and the MB$^+$-tree also use linear orders such as Z order and column-wise order. However, they have a shortcoming in common with respect to the spatial locality (see Fig. 2). The MB$^+$-tree divides the data space into three regions M1, M2 and M3. This may distribute objects such that the distant objects are clustered in the same region instead of nearby objects. The zkdb tree and G-tree have the same problem as shown in Fig. 2(b). On the other hand, the directory regions, H1, H2 and H3 of the HG-tree in Fig. 2(c) are compact and the nearby objects are clustered in the same region.

The Hilbert R-tree [14] also uses a Hilbert ordering scheme, however it is one for spatial data not point data and its directory regions are rectangular and overlapping unlike the HG-tree. The HG-tree improves performance through more general shape of directory region. Moreover, since its directory regions are disjoint the insertion, deletion, and exact match search is restricted to exactly only one path.



(a) MB$^+$-tree (b) zkdb tree (c) HG-tree

Fig. 2. The points in a 2-dimensional space are arranged in (a) column-wise scan order, (b) Z order, and (c) Hilbert order. The dashed lines shows the ordering path and the heavy lines show the partitioning boundaries. The squares in grid cells represent data points.

1. *Basic Ideas and Properties*

The main idea of the HG-tree is to create an indexing scheme that it can support the followings:

1. when an overflow occurs in a node try to absorb it and when splitting is necessary convert two nodes to three nodes. As a result of this behavior the index structure has higher storage utilization;
2. maintain the directory region in a minimal way to reduce the directory coverage;
3. control the correlation between the storage utilization and the directory coverage to compromise the trade-off between them;
4. maintain the node occupancy not to be fallen below a certain minimum to have predictable and controllable worst- case characteristics.

To absorb splitting we need the ordered list of the objects. We transform Cartesian coordinates in an *n*-dimensional feature space into locations on the Hilbert curve. Thus, objects are represented by locations on the Hilbert curve and we can store them in a sorted order. Through this ordering every node has a well-defined set of siblings and the HG-tree absorbs splitting by redistributing the objects of the

overflowing node into adjacent sibling and adjusting the directory regions. When splitting is necessary, convert the two nodes, the overflowing node and one of the two adjacent siblings, to three nodes (see Fig. 3). Thus this splitting is called *2-to-3* splitting. When we select one of the two adjacent siblings, we select one that makes the directory coverage minimal. Resulting from this, the HG-tree yields average storage utilization more than 80% and guarantees the worst-cast storage utilization is more than 66.7% (2/3) of full capacity. This concept is similar to that used with the B*-tree [3]. However, the B*-tree operates on a 1-dimensional space and the HG-tree can be viewed as a generalization of it.

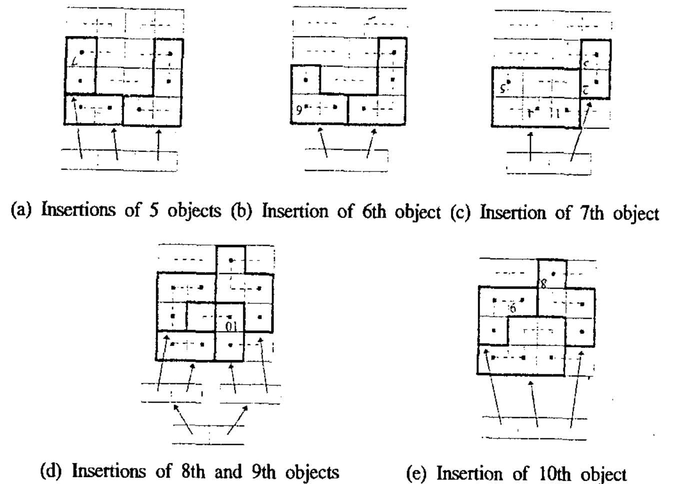
To reduce the directory coverage we introduce the concept of *minimum bounding interval* (*MBI*) that covers all regions of the lower nodes. This plays a similar role as a *minimum bounding rectangle* (*MBR*) used in the SAM such as R-tree. But it does not allow overlap and is not rectangular. It is used to determine whether or not a subtree rooted with one of the children will be visited during searching and data insertion. For every internal node of the HG-tree, its MBI is stored. Specifically, an internal node in the HG-tree contains at most C_n entries of the form

$$(I, ptr)$$

where C_n is the capacity of an internal node, I is the MBI that encloses all the children of that entry and that is represented by two Hilbert values at either end of the interval, and *ptr* is a pointer to the child node. We maintain these entries in a Hilbert order. Another advantage of using linear order is that binary search can be used in searching these entries within a node. When a node is large, the difference between a binary search method, with a $log(n)$ cost and linear search, with an average cost of $n/2$ is significant, where n is the number of entries in a node.

With the example 1 we intend to visualize the basic properties of the HG-tree:

Example 1. Let the dimension be 2 and both of the capacities of a directory page and a data page be 3. The snapshots in Fig. 3 depict the growth of the HG-tree. Each grid cell corresponds to a point of the space, each square and the number beside it represent a data object and the sequence of it inserted, respectively, and the dotted line depicts the Hilbert curve. In the data pages the actual objects are stored. The MBIs having at most 3 objects are depicted by a fill pattern. The white area correnspońds to empty data space not managed by the HG-tree. When the 6th object is inserted, the data page is not split but the objects are distributed into its adjacent sibling and the directory regions are adjusted as in Fig. 3(b). When the 7th object is inserted, two nodes are split into three as in Fig. 3(c). The insertion of the 10th object makes an overflow of data page and in turn directory page increasing the height of HG-tree.



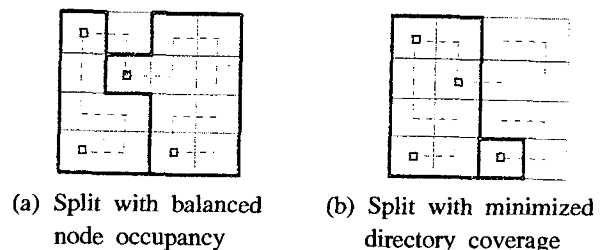
(a) Insertions of 5 objects (b) Insertion of 6th object (c) Insertion of 7th object

(d) Insertions of 8th and 9th objects (e) Insertion of 10th object

Fig. 3. The sequence of insertions of 10 objects on the G-tree.

Although it is known that the trees with best storage utilization may produce nearly best query performance [24], it is not always the case. Depending on the data distribution, the minimal property of directory region can play more important role than the maximal storage utilization in query performance. From the extensive performance tests we have got the experience that there is a trade-off between the maximal storage utilization and the minimal directory coverage. When we increase the storage utilization by absorbing splitting, there is also a tendency to increase the directory coverage. On the other hand, the storage utilization is reduced if we only intend to reduce the directory coverage. With this insight we can relax the 2/3 minimum node occupancy resulting from 2-to-3 splitting to some extent to reduce the directory coverage. In HG-tree, we can get good search performance regardless of data distibution by controlling the two system parameters, the storage utilization and the directory coverage adaptively depending on data distribution.

Example 2. Fig. 4(a) shows the two nodes resulting from the split with balanced node occupancy. While this splitting may enhance the storage utilization, it may result in high directory coverage. This may have a negative effect on the



(a) Split with balanced node occupancy (b) Split with minimized directory coverage

Fig. 4. Two node-split policies.

query performance depending on data distribution. In such a case, the insertion algorithm of the HG-tree controls the node occupancy to reduce the directory coverage at the expense of the balanced node occupancy. From this, we can get the result of Fig. 4(b). The directory coverages of Fig. 4(a) and Fig. 4(b) are 15 and 9, respectively.

2. *Insertion and Deletion*

To insert a new object in the HG-tree, we first calculate the Hilbert value *h* of the object and traverse the tree using it as a key. In each level we choose the branch with minimum Hilbert distance from *h* among the entries in the node. Once we reach the leaf level, we insert the object in its correct order according to *h*. Overflow is handled by redistributing some of its contents to one of its adjacent siblings or splitting 2 nodes into 3. After a new object is inserted, we update the MBIs of the affected nodes along the path. The redistribution algorithm tries to make the directory coverage minimum.

A deletion is straightforward unless it causes an underflow. In such a case, an underflowing node resulting from a deletion can borrow keys from or merge with its its adjacent siblings. So the algorithms for deletion are omitted.

Algorithm Insert(node *N*, object *o*)

```
/* Insert object o into tree rooted at N. h is the Hilbert value of the object */
{
  I1. [Find position for new object o]
      Use ChooseLeaf(o, h) to choose a leaf node L in which to place o.
  I2. [Add object o to leaf node L]
      Insert o into L in the appropriate place according to the Hilbert order.
      If L overflows, invoke HandleOverflow(L, o), which will return new leaf
      if split was inevitable.
  I3. [Propagate changes upward]
      Form a set S that contains L, its adjacent sibling, and the new leaf (if any).
      Use AdjustTree(S) to update the MBIs that have been changed.
  I4. [Grow tree taller]
      If node split propagation caused the root to split, create a new root whose
          children are the two resulting nodes.
}
```

Algorithm ChooseLeaf(object *o*, long int *h*)

```
/* Select a leaf node in which to place a new object o */
{
  C1. [Initialize] Set N to be the root node.
  C2. [Leaf check] If N is a leaf, return N.
  C3. [Choose subtree]
      If N is not a leaf, choose the entry (I, ptr) with the minimum Hilbert
          distance from h to I.
  C4. [Descend until a leaf is reached]
      Set N to the node pointed by ptr and repeat from C2.
}
```

Algorithm AdjustTree(set *S*)

```
/* S is a set of nodes that contains the node L being updated, its adjacent sibling
(if overflow has occurred), and newly created node NN (if split has occurred).
The routine ascends from leaf level towards the root, adjusting MBI of nodes that
cover the nodes in S. It propagates splits (if any) */
{
  AO1. [Initialize] Set N=L.
  AO2. [Check if done] If N is the root, stop.
  AO3. [Propagate node split upward]
       Let P be the parent node of N.
       If N has been split, let NN be the new node.
          Insert NN in P in the correct order according to its Hilbert value if
          there is room.
          Otherwise, invoke HandleOverflow(P,NN).
       If P is split, let PP be the new node.
  AO4. [Adjust the MBIs in parent entry]
       Let Z be the set of parent entries for the nodes in S.
       Adjust the corresponding MBIs of the entries in Z appropriately.
  AO5. [Move up to next level]
       Repeat from AO2 with N = P and NN = PP, if P was split.
}
```

Algorithm HandleOverflow(node *N*, object *o*)

```
/* Return the new node if a split occurred */
{
  HO1. Let S be a set that contains all the entries from N and its adjacent sibling.
  HO2. Add o to S.
  HO3. If at least one of the 2 siblings is not full,
          distribute S between N and its adjacent sibling.
  HO4. If all the 2 siblings are full,
          create a new node NN and distribute S among the nodes N, NN, and
       its adjacent sibling.
       return NN.
}
```

3. *Range Searching*

The search algorithm starts with the root and examines each branch that intersects the query region recursively following these branches. At the leaf level it reports all entries that intersect the query region as qualified objects. Examining for intersection, the algorithm first expands the MBI of the node into the MBR that covers the MBI. If the MBR is fully contained in the query region, then all objects in the node satisfy the query, or else if it is outside the query region, it does not have to be considered any further. On the other hand, if it overlaps, the algorithm divides the original MBI into two sub-MBIs, MBI_1 and MBI_2 along the point bisecting the MBR according to Hilbert sequence and computes their MBRs, MBR_1 and MBR_2 (see Fig. 5). With these two new MBRs, the algorithm re-examines for intersection. This process proceeds recursively until the sum of the areas of the resulting MBRs is equal to the area of the MBI or the MBI is known to be fully contained in or outside the query region. At this point it is determied whether the MBI overlaps, is wholly contained, or outside the query region. For example, in the first test for intersection in Fig. 5(a), the MBR overlaps the query region. Since the areas of the MBR and the MBI are not equal (the areas of MBR and MBI are 8 and 6, respectively), the MBR

is bisected into MBR1 and MBR2 as shown in Fig. 5(b). It then compares them with the query region, saparately. At this point, HG-tree determines that the directory region is outside the query region.

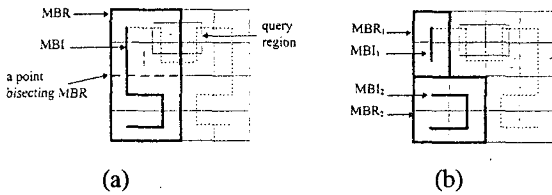


Fig. 5. Test for intersection in range query processing.

Algorithm RangeSearch(node *N*, QueryRegion *r*)

/ Perform a query with range r, Ni.child is a child node of node Ni */*

```
{
  R1. [Search nonleaf node]
        For every entry Ni in N
            Invoke Overlap(MBR(Ni), r) to determine it is contained in,
            overlaps, or outside r.
            If it is contained in, then output all objects belonging to Ni.
            Else if it overlaps, then invoke RangeSearch(Ni.child, r).
  R2. [Search leaf node]
        Output all the objects that intersect r.
}
```

Algorithm Overlap(MBR *I*, QueryRegion *r*)

/ Determine the MBR I is contained in, overlaps, or outside the query region r */*

```
{
  O1. If I is contained in or outside r, return its corresponding indicator.
        Else if the area of I is equal the area of MBI, return an indicator that
        it overlaps r.
        Else
            Bisect the MBI into two MBIs, I1 and I2, along the mid-point of
            the MBR.
            Invoke Overlap(MBR(I1), r) and Overlap(MBR(I2), r)
            If both of them return indicators that they are contained in or
            outside,
        return its corresponding indicator.
            Else return an indicator that it overlaps r
}
```

4. *Nearest-Neighbor Searching*

Let D be a distance function defined on an n-dimensional space. Given a point $(x_1, x_2, ..., x_n)$ and a positive integer k, the k-nearest neighbor query finds the k nearest neighbors of $(x_1, x_2, ..., x_n)$ with respect to the distance function D.

We can use a branch-and-bound algorithm [25] for k-nearest neighbor queries on the HG-tree. Two lower and upper distance value bounds δ_{low} and δ_{high} are introduced to order and efficiently prune the paths of the search space in the HG-tree:

- δ_{low} gives the minimum distance between a given point and an MBI;
- δ_{high} gives the distance between a given point and an MBI that guarantees the finding of an object in MBI at a Euclidean distance less than or equal to this distance.

δ_{low} is used to determine the closest object to a given point from all those in MBI. However, in fact, due to empty space inside the MBIs, the nearest neighbor might be much further than δ_{low}. δ_{high} guarantees the presence of an object in MBI within this distance, because there must be objects in the both ends of the MBI. We use these bounds for ordering and pruning the search tree.

Definition 3. *Given a point P in Euclidean space of dimension n and an MBI I = (H_1, H_2), where H_1 and H_2 are the two end points on I, we define $\delta_{high}(P, I)$ as:*

$$\delta_{high}(P, I) = \min(D(P, H_1), D(P, H_2))$$

where D is an Euclidean distance Function.

Note that computing δ_{high} requires only two comparisons. The computation of δ_{low} is performed recursively. First of all, we get the MBR that covers the MBI and compute the minimum distance of a point in MBR from the query point. If this point lies in the MBI, then δ_{low} is the computed distance. Otherwise, we bisect the MBI into two sub-MBIs as in the way used in the intersection test of the range search and perform above procedure recursively with the two sub-MBIs. Finally, the minimum of the minimum distances computed with two sub-MBIs is determined as δ_{low}. Fig. 6 illustrates low and high in a 2-dimensional space. Each grid cell corresponds to a point in a domain space.

The algorithm works as follows : given a query point, examine the top-level branches, compute the lower bound, δ_{low} and the upper bound, δ_{high} for the distance, and traverse the most promising branch with the depth first order. At each stage of traversal, the order of search is determined by the nondecreasing order of δ_{low}. The objects with the distance to a given query point greater than δ_{high} of the farthest MBI and the MBIs with δ_{low} greater than the distance between the query point and the farthest object are discarded in each traversal stage.

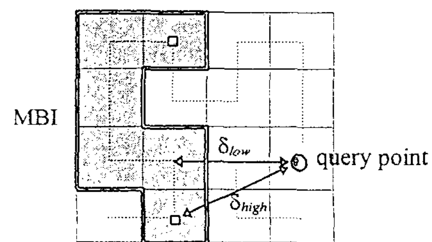


Fig. 6. δ_{low} and δ_{high} in a 2-dimensional space.

Algorithm NNSearch(Node *N*, Point *P*, NearestNeighbor *Nearest*)

/ Return the k nearest neighbors. N is a current node, P is a search point, and Nearest is the sorted list holding the k current nearest neighbors in nondecreasing distance order. BranchList is a list holding branches of nonleaf node. Ei.child is a child node of node Ei */*

```
{
  N1. [Search leaf node]
      For every entry Ni in N
      Determine the distance, disti, between query point P and the object Ni.
      If disti is less than Nearest[k].dist
        Assign disti to the Nearest[k].dist.
        Assign Ni to the Nearest[k].obj.
        Rearrange Nearest in correct order.
  N2. [Search Nonleaf Node]
      For every entry Ni in N
        Compute the metric bounds δlow and δhigh and store it with Ni into a
        BranchList.
      Sort the BranchList based on δlow.
      Remove the unnecessary branches in BranchList as compared with
      Nearest.
      For arranged entries Ei in BranchList
         Invoke NNSearch(Ei.child, P, Nearest)
         Remove the unnecessary branches in BranchList as compared with
         Nearest.
}
```

IV. Experimental Results and Analysis

To assess the performance of the HG-tree, we implemented it in C under MS-DOS and ran experiments on a four dimensional space using a pentium PC. We compared our method against the buddy-tree. Seeger and Kriegel [26] reported that the buddy-tree is the best one among multidimensional PAMs with respect to the average range query performance. For all operations, we have measured the number of disk accesses per operation.

1. *Experimental Setup*

To experiment we generated five groups of four dimensional data files that contained different distributions of data as in [26]: uniform, diagonal, bit, x-parallel, and clustered distributions. Each file contains 100,000 objects without duplicates. To demonstrate the performance for range queries we generated six groups of range queries. The regions of the six groups are squares varying in size which are 0.01%, 0.1%, 1%, 10%, 20%, and 40% of the data space and their centers are uniformly distributed in the data space. To test the nearest-neighbor queries, the numbers of neighbors we used are 20, 40, 60, 80 100, and 120. For each experiment, 1,000 randomly generated queries were asked and the results were averaged. The page size used for data pages and directory pages is 512 bytes.

As the storage utilization is controllable in the HG-tree, we experimented two types of the HG-tree with minimum storage utilization of 66.7% (2/3) and 25% (1/4), which are abbreviated by HG* and HG+, respectively, in the results. The buddy-tree is abbreviated by BUDDY.

2. *Results and Analysis*

For the queries with ranges 0.01% to 40%, we report the average number of disk accesses per query in Fig. 7. Fig. 8 shows the average of 1,000 nearest neighbor queries for each of several different number of nearest neighbors. Tables 2, 3, and 4 show the range query and nearest neighbor query, and insertion costs for each distribution as averages over all five types of queries, respectively. For the sake of an easier comparability, we have normalized the average number of page accesses for the queries and the average index file size in BUDDY to 100% in each table. In Table 5 we computed the unweighted average over all five distributions.

In Table 5 we measured the following parameters:

- ○ range query: the unweighted range query average over all five groups of data files
- ○ n. n. query: the unweighted nearest neighbor query average over all five groups of data files
- ○ storage utilization: the average storage utilization
- ○ file size: the average index file size
- ○ insert: the average number of page accesses for an insertion

1) *Search Cost*

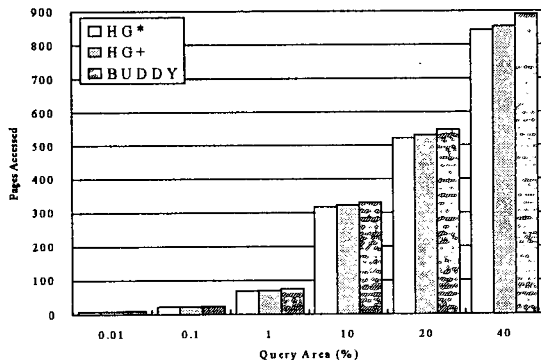
Considering Table 5, the HG-tree offers itself to be the winner of our comparison. We have to take a closer look at the different distributions. In all cases of the Fig. 7, with the exception of the small ranges on the clustered distribution, the HG-tree outperforms the buddy-tree in all range query performance. For small range queries on the clustered distribution, the directory coverage is a more important factor than the storage utilization. The HG+ reduces the directory coverage by reducing the storage utilization to at least 25% (1/4). The performance of range queries of the HG-tree on the clustered distribution improves as the size of the query range goes beyond 1%. The superiority of the HG-tree to the buddy-tree comes from high storage utilization above 80% and the control over the node occupancy. For the nearest-neighbor queries, shown in the Fig. 8, the HG-tree outperforms the buddy-tree with the exceptions of the x-parallel distribution and the queries with the small number of neighbors on the clustered distribution. The main observation of the Table. 5 is that there is no clear winner in the nearest neighbor query.

2) *Insertion Cost and Storage Utilization*

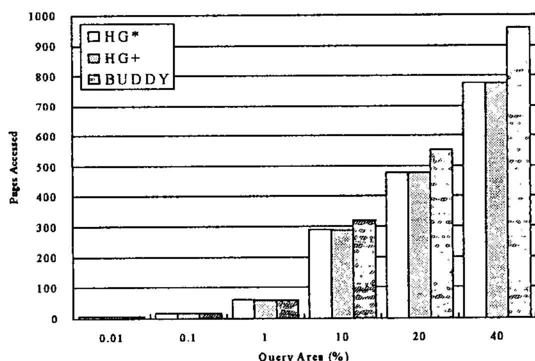
We measured the number of disk accesses (read+write) needed to build the indexes. We assumed that every update of the index would be reflected on the disk. Since the HG-tree tries to absorb splitting and employs the 2-to-3 splitting, the number of nodes need to be inspected at overflow increases. However, Table 4 shows that there is no clear winner in the insertion cost.

3) Storage Requirements

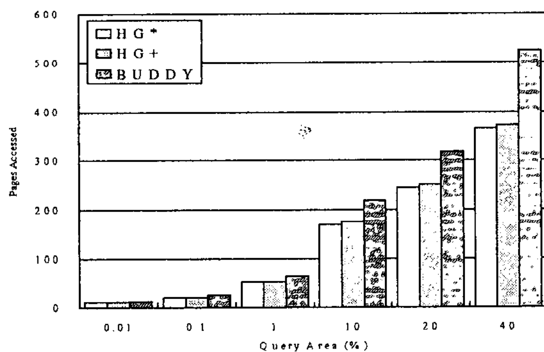
The HG-tree requires fewer number of nodes (and thus less storage) than the buddy-tree. The savings are around 30%.



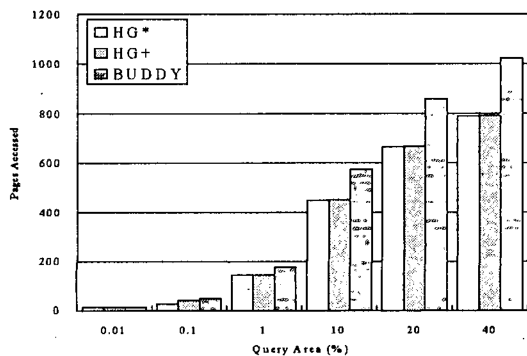
(a) Uniform distribution



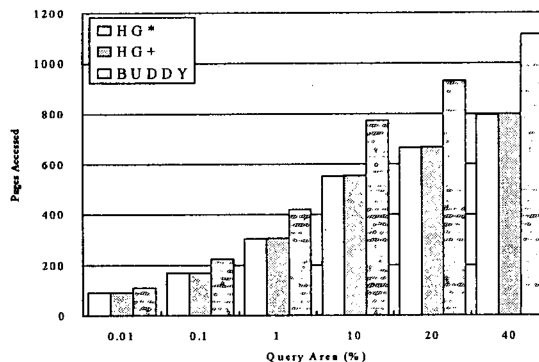
(b) Clustered distribution



(c) Bit distribution



(d) x-parallel distribution

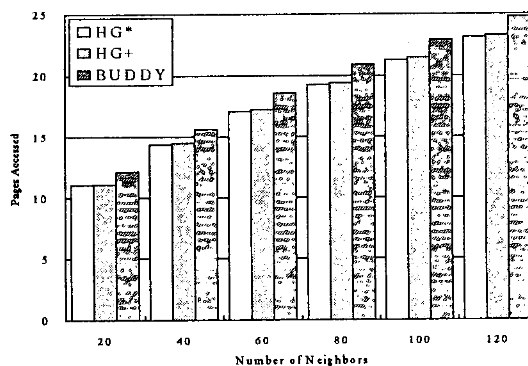


(e) Diagonal distribution

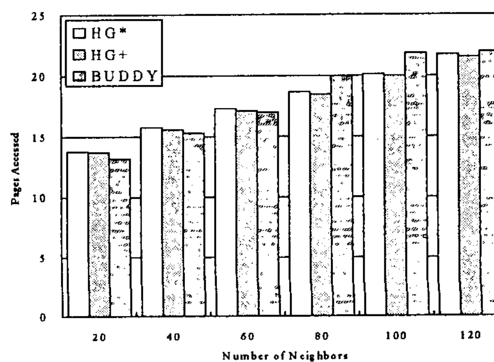
Fig. 7. Range Query Performance.

Table 2. Unweighted average over all 5 types of range queries depending on the distribution.

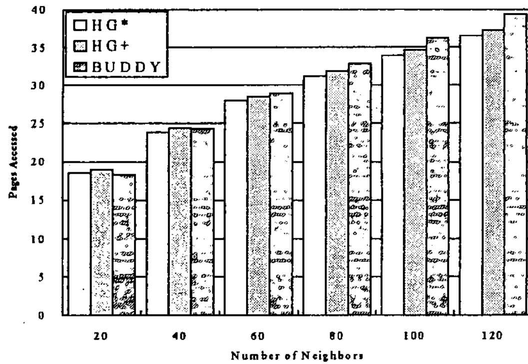
	uniform	clustered	bit	x-parallel	diagonal
BUDDY	100.0	100.0	100.0	100.0	100.0
HG*	94.8	85.1	74.3	77.5	74.3
HG+	96.3	84.3	75.8	78.4	74.5



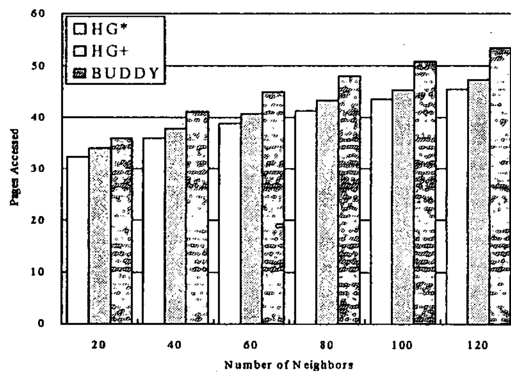
(a) Uniform distribution



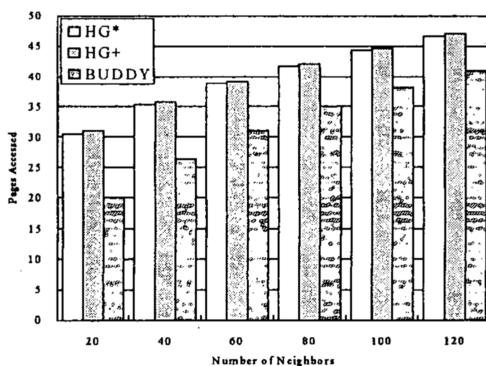
(b) Clustered distribution



(c) Bit distribution



(d) x-parallel distribution



(e) Diagonal distribution

Fig. 8. Nearest Neighbor Query Performance.

Table 3. Unweighted average over all 5 types of nearest neighbor queries depending on the distribution.

	uniform	clustered	bit	x-parallel	diagonal
BUDDY	100.0	100.0	100.0	100.0	100.0
HG*	92.1	98.4	95.6	124.0	86.5
HG+	92.9	97.4	97.6	125.2	90.6

Table 4. unweighted average over all 5 types of inserts depending on the distribution.

	uniform	clustered	bit	x-parallel	diagonal
BUDDY	4.07	4.70	5.20	5.34	5.24
HG*	4.09	4.73	3.51	3.97	4.09
HG+	4.67	4.75	4.09	4.59	4.72

Table 5. unweighted average over all 5 distributions.

	range query	n.n. query	storage utilization	file size	insert
BUDDY	100.0	100.0	61.8	100.0	4.91
HG*	81.2	99.3	84.3	69.4	4.08
HG+	81.9	100.7	80.7	72.7	4.56

V. Conclusions

In this paper, we proposed the HG-tree as an access method for content-based retrieval in large image database. Contrary to previously suggested multidimensional PAMs, the HG-tree tries to absorb splitting and when splitting is necessary, converts two nodes to three by using the Hilbert ordering. Through these properties, the HG-tree achieves the average storage utilization more than 80%. Moreover, it attempts to reduce the directory coverage by maintaining the directory regions as minimal as possible. In addition, by controlling the trade-off between the maximal storage utilization and the minimal directory coverage it can cope with a wide range of data distributions.

Based on these ideas, we designed and implemented the HG-tree and carried out performance experiments, comparing our method to the buddy-tree. Summarizing the outcome of our comparisons, we can state that the HG-tree exhibits around 18% better average range query performance than the buddy-tree, and results in a reduction in the size of a tree, and hence its storage cost. The good performance of the HG-tree is not by chance, but comes from the overall control over the storage utilization and the directory coverage.

Future work could focus on the analysis of the PAM including the HG-tree, providing analytical formulas that pridict the response time of the nearest neighbor queries. Another area of future research is the integration of the HG-tree with different types of indexing methods such as signature files [8], iconic indexes [2, 4] to support a wide range of queries, both text-based and content-based.

Acknowledgment

This research was supported by the Fundamental Research on Multimedia Engines Project sponsored by the Samsung Group.

References

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles," *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, pp. 322-331, 1990.

[2] A. D. Bimbo, M. Campanai, and P. Nesi, "A Three-Dimensional Iconic Environment for Image Database Querying," *IEEE Transactions on Software Engineering*, Vol. 19, No.10, October, 1993.

[3] D. Comer, The Ubiquitous B-tree, *ACM Computing Surveys*, Vol. 11, No. 2, pp. 121-137, 1979.

[4] G. Costagliola, G. Tortora, and T. Arndt, "A Unifying Approach to Iconic Indexing for 2-D and 3-D Scenes," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 3, June, 1992.

[5] C. Faloutsos, "Gray codes for partial match and range queries," *IEEE Transactions on Software Engineering*, Vol. 14, No.10, pp. 1381-1393, 1988.

[6] C. Faloutsos and I. Kamel, "Beyond Uniformity and Independence: Analysis of R-trees Using the Concept of Fractal Dimension," *Proceedings of SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 4-13, 1994.

[7] C. Faloutsos and S. Roseman, "Fractals for secondary key retrieval," *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 247-252, 1989.

[8] W. B. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithm*, Prentice Hall, 1992

[9] M. Freeston, "The BANG File: a New Kind of Grid File," *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 260-269, 1987.

[10] M. Freeston, "A General Solution of the n-dimensional B-tree Problem," *Proceedings of ACM SIGMOD Internatinal Conference on Management of Data*, pp. 80-91, 1995.

[11] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 47-57, 1984.

[12] D. Hilbert, Uber die stetige Abbildung einer Linie auf ein Flachenstuck, *Math. Annalen*, Vol. 38, 1891.

[13] H. V. Jagadish, "Linear Clustering of Objects with Multiple Attributes," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 332-342, 1990.

[14] I. Kamel and C. Faloutsos, "Hilbert R-tree: An improved R-tree using fractals," *Proceedings of the 20th VLDB Conference*, pp. 500-509, 1994.

[15] A. Kumar, G-Tree: A New Data Structure for Organizing Multidimensional Data, *IEEE Tansactions on Knowledge and Data Engineering*, Vol. 6, No. 2, pp. 341-347, 1994.

[16] D. B. Lomet and B. Salzberg "The hB-tree: a Robust Multi-Attribute Indexing Method," *ACM Transactions on Database Systems*, Vol. 15, No. 4, pp. 625-658, 1990.

[17] 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," *Proc. of the SPIE Conf. on Storage and Retrieval for Image and Video Databases*, pp. 173-187, 1993.

[18] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The grid file: an adaptable, symmetric multikey file structure," *ACM Transactions on Database Systems*, Vol. 9, No.1, pp. 38-71, 1984.

[19] J. A. Orenstein and T.H. Merrett, "A Class of Data Structures for Associative Searching," *Proc. of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pp. 181-190, 1984.

[20] E. J. Otoo, "Balanced multidimensional extendible hash tree," *Proc. of the 5th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 100-113, 1986.

[21] B. W. Pagel, H. W. Six, H. Toben, P. Widmayer, "Towards an Analysis of Range Query Performance in Spatial Data Structures," *Proc. of the 12h ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 214-221, 1993.

[22] G. Peano, "Sur une courbe qui remplit toute une aire plane," *Math. Annalen*, Vol. 36, pp. 157-160, 1890.

[23] J. T. Robinson, "The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes," *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pp. 10-18, 1981.

[24] A. L. Rosenberg and L. Snyder, "Time- and Space-Optimality in B-Trees," *ACM Transactions on Database Systems*, Vol. 6, No. 1, pp. 174-183, 1981.

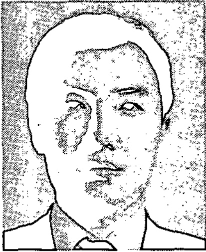
[25] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest Neighbor Queries," *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pp. 71-79, 1995.

[26] B. Seeger and H. P. Kriegel, "The Buddy-Tree: An Efficient and Robust Access Method for Spatial Data

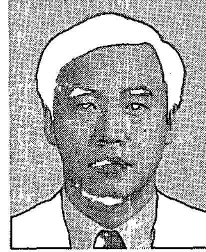
Base Systems," *Proceedings of the 16th VLDB Conference*, pp. 590-601, 1990.

[27] K. Y. Whang, S.-W. Kim, and G. Wiederhold, "Dynamic Maintenance of Data Distribution for Selectivity Estimation," *VLDB Journal*, Vol. 3, No.1, pp. 29-51, 1994.

[28] Q. Yang, A. Vellaikal, and S. Dao, "MB$^+$-Tree: A New Index Structure for Multimedia Databases," *Proc. of the International Workshop on Multi-Media Database*, pp. 151- 158, 1995.



Guang-Ho Cha received the B.S. degree in computer science & statistics from Pusan National University in 1984 and the M.S. degree in computer science from Korea Advanced Institute of Science and Technology in 1989. He was an software engineer at Samsung Electronics and is currently employed by DACOM corp. as a senior software engineer, and is pursuing a Ph.D. in the department of information & communication engineering at KAIST. His current research interests include multimedia software and systems, object-oriented database systems, computer graphics, and image processing.



Chin-Wan Chung received the B.S. degree in electrical engineering from Seoul National University in 1973 and the Ph.D. degree in computer engineering from the University of Michigan in 1983. From 1983 to 1993, he was a senior research scientist and a staff research scientist in the Computer Science Department at General Motors Research Laboratories. Since 1993, he has been an associate professor in the Department of Information and Communication Engineering at Korea Advanced Institute of Science and Technology, Seoul Campus. He is a member of the Korea Information Science Society, IEEE Computer Society and ACM. His research interests include object-oriented databases, geographic information systems, multimedia databases, distributed databases, and CIM.