An Efficient Content-Based High-Dimensional Index Structure for Image Data

Jang Sun Lee[a], Jae Soo Yoo, Seok Hee Lee, and Myung-Joon Kim

The existing multi-dimensional index structures are not adequate for indexing higher-dimensional data sets. Although conceptually they can be extended to higher dimensionalities, they usually require time and space that grow exponentially with the dimensionality. In this paper, we analyze the existing index structures and derive some requirements of an index structure for content-based image retrieval. We also propose a new structure, for indexing large amount of point data in a high-dimensional space that satisfies the requirements. In order to justify the performance of the proposed structure, we compare the proposed structure with the existing index structures in various environments. We show, through experiments, that our proposed structure outperforms the existing structures in terms of retrieval time and storage overhead.

Manuscript received July 15, 1999; revised May 17, 2000.

[a] Electronic mail: sunny@computer.etri.re.kr

I. INTRODUCTION

Many recent applications such as image databases, medical databases, Geographic Information Systems (GIS) and CAD/CAM applications require enhanced indexing for content-based image retrieval [1]–[6]. Content-based image retrieval is used to query large on-line databases using the content of images (such as color, texture, and shape of image objects) as the basis of queries [7]–[10]. In applications that require content-based retrieval, indexing of high-dimensional data has become increasingly important for fast retrieval. For example, in image databases, image objects are usually mapped into feature vectors in some very high-dimensional space. Queries are processed on a database that indexes the feature vectors. Therefore, index structures for high-dimensional data are required to efficiently support content-based retrieval.

There are several index structures for high dimensional data such as SS-tree [11], [12], TV-tree [13], X-tree [14] and SR-tree [15]. The SS-tree was proposed as an index structure to efficiently support similarity searches. The idea of the TV-tree comes from the observation that, in most high-dimensional data sets, a small number of the dimensions bears most of the information. The main idea of the X-tree is to minimize the overlap of bounding boxes in the directory by using a new organization of the directory. The SR-tree is an extension of the R*-tree [16] and the SS-tree. The SR-tree uses both bounding spheres and bounding rectangles to improve the performance on the nearest neighbor queries [17]. Conceptually they can be extended to higher dimensionalities; however, they are not suitable for an index structure for content-based retrieval, because they usually require time and space that grow exponentially with the increase in the dimensionality [18]. The Pyramid-Technique was proposed based on a special partitioning strat-

egy to break the so-called *curse of dimensionality* [19]. It is suitable for high-dimensional range queries.

In this paper, we derive some design requirements of an index structure for content-based image retrieval. We also propose a new index structure, called CIR (Content-based Image Retrieval)-tree, for indexing large amounts of point data in a high-dimensional space that satisfies the derived requirements. We perform extensive experiments with real data as well as synthetic data. The relationships among various performance parameters are thoroughly investigated. Through these experiments, we show that the CIR-tree significantly improves performance in terms of the retrieval time and the storage overhead over TV-tree, X-tree and Pyramid-Technique.

The remainder of this paper is organized as follows. In Section II, we describe related work. In Section III, we present a few design requirements of an index structure for content-based retrieval. In Section IV, we propose a new index structure that satisfies the requirements. We evaluate the performance of the index structure with real and synthetic data sets and describe the evaluation results in Section V. Finally, conclusions are described in Section VI.

II. RELATED WORK

The R-tree [20] and its most successful variant, the R*-tree, have been used most often for indexing high dimensional data in database literature. The R-tree is a height-balanced tree corresponding to the hierarchy of nested rectangles. The rectangle of an internal node is determined by the minimum-bounding rectangle that covers all the rectangles of its children. The rectangle of a leaf node is determined by the minimum bounding rectangle that covers all data entries in that leaf node. Therefore, the rectangle of the root node corresponds to the minimum bounding rectangle of the whole data entries.

The R*-tree has two major enhancements over the R-tree. First, rather than considering the area only, it minimizes the perimeter and overlap of each enclosing rectangle in the internal nodes. Second, the R*-tree introduces the notion of forced reinsertion to make the shape of the tree less dependent on the order of insertions. However, the R-tree and the R*-tree explode exponentially with the increase of dimensionality, eventually being reduced to a sequential scanning.

Another R-tree like index structure is the SS-tree, which uses k-dimensional spheres as minimum bounding regions instead of rectangles [11]. The SS-tree provides better performance than the R*-tree. However, spheres tend to overlap in high-dimensional spaces, because the volume of bounding spheres is much larger than that of bounding rectangles. To avoid the overlap problem, the SR-tree combines the concepts of the R-tree and SS-tree [15]; it outperforms both the R*-tree and the SS-tree.

The basic idea of the Pyramid-Technique is to transform the d-dimensional data points into 1-dimensional values [19]. For storing and accessing the values, it uses the B^+-tree [21]. Potentially, any order-preserving one-dimensional access method can be used. Operations such as insert, update, delete or search are performed by using the properties of B^+-tree.

The TV-tree is a method in the database literature that was proposed specifically for indexing high-dimensional data [13]. The basis of the TV-tree is to use contracting and extending dynamically feature vectors. That is, it uses as little features as possible that are necessary to discriminate between objects. An example of a TV-tree is given in Fig. 1. The points denoted from A to I represent data points (only the first two dimensions are shown). In the root level, region R3 uses only one dimension for discrimination and other regions use two dimensions for discrimination. The TV-tree, however, does not consider the overlap problems and thus it still has the possibility of rapid performance degradation in processing queries.

The X-tree was proposed as an index structure to avoid splits that would result in high degree of overlaps in the directory [14]. To do this, the X-tree uses a split algorithm minimizing overlaps and additionally uses the concept of supernodes. Supernodes are large directory nodes of variable size (a multiple of the usual block size). Supernodes are created during insertion to avoid splits in the directory that would result in a highly overlapped structure. The X-tree uses the notion of maximum overlap value (*MaxO*) to decide whether it splits a node or extends a node to a supernode. Most insertion algorithms split a node in two if an overflow occurs. If the possibility of the overlap of the two split nodes becomes larger than *MaxO*, the X-tree extends the original node into a supernode instead of splitting it. The suggested value of *MaxO* in [14] is 20%.

Due to the fact that overlaps increase as the number of dimensions increases, the number and size of supernodes increase in the X-tree [14]. Figure 2 shows three examples of X-tree with different dimensionalities.

Although the overlap is reduced in the directory, the X-tree loses the efficiency of hierarchical structures. In Fig. 2, when the number of dimensions D is 32, the structure of the X-tree looks linear because of large supernodes.

To solve the overlap problems of the TV tree, our proposed CIR-tree will suggest an improved ChooseSubtree algorithm that chooses the most appropriate node for inserting an image object. The detailed algorithm is described in Section IV-3.A. The CIR-tree also adapts the supernode concept in the Split algorithm to alleviate the overlap problem. However, because our CIR-tree uses smaller feature vector than the X-tree in the directory, the size of supernodes and the total size of the directory will decrease.

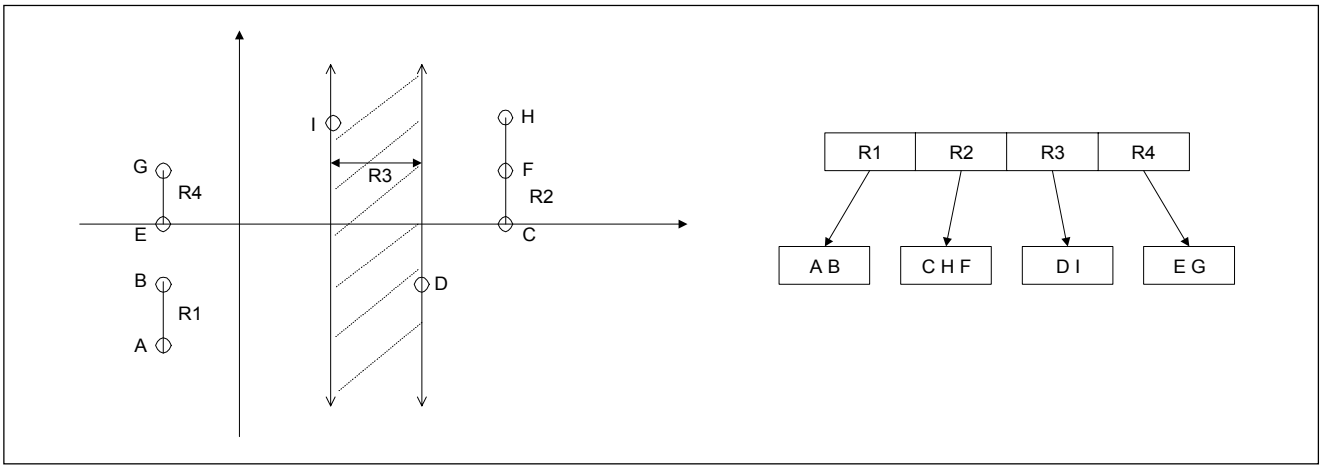


Fig. 1. An example of TV-tree. In the root level, region R3 uses only one dimension for discrimination. But other regions use two dimensions for discrimination.

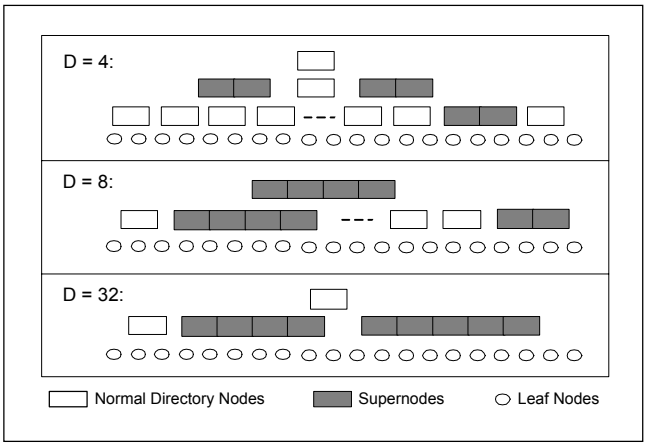


Fig. 2. Various shapes of the X-tree in different dimensions.

III. DESIGN REQUIREMENTS FOR HIGH DIMENSIONAL INDEX STRUCTURES

From the survey of the existing high dimensional index structures and the characteristics of content-based retrieval, we believe that a newly created index structure for high-dimensional data should be designed by considering the following requirements:

• *An index structure must efficiently deal with high dimensional features.*

Index structures for a content-based image retrieval system must deal with high dimensional image features. Most existing multi-dimensional index structures are not adequate for handling high dimensionality: they become extremely inefficient with growing dimensionality because the number of nodes increases exponentially. When an index structure is constructed, the number of nodes should not increase exponentially as the number of dimension increases.

• *The overlap between directory regions must be minimized.*

In general, an overlap means a region that is covered by more than one directory area. As the amount of data and the height of a tree increase, the overlap area increases remarkably with the growing dimensionality of data. Usually, since the overlap results in the necessity of traversing multiple paths, it badly affects the performance of processing queries. Consequently a newly created index structure should provide an algorithm to minimize overlaps.

• *Storage utilization must be optimized.*

Higher storage utilization generally reduces the query cost since the height of the tree would be kept low. Eventually, some query types like a range query that finds all objects that are within a certain distance from a given query object are more likely to be influenced by the storage utilization, because the objects may be spread across more nodes as the storage utilization becomes low.

• *An index structure must be appropriate for similarity-based queries.*

Unlike conventional database systems, a content-based image retrieval system processes queries based on similarity, since images are not atomic symbols but unformatted data [22]. Therefore the index structure must process similarity queries efficiently.

• *An index structure must employ a similarity measure.*

Image features are expressed as points in a high-dimensional feature space in most content-based image retrieval systems. Euclidean distance between two points is generally used as a similarity measure in the systems. However, measuring the

similarity between two points in a feature space with just Euclidean distance may not match the user's notion of similarity, since the dimensions of image features are independent of one another and different in respect of relativity and distribution. As a result, other similarity measures must be employed.

- *The index structure must process various query types efficiently.*

An index structure has to be able to process various query types such as an exact match query, a partial match query, a range query and a k-nearest neighbor search query. An index structure must achieve good performance on every query.

- *An index structure has to deal with high-dimensional features dynamically.*

Though there are certain applications having an archival nature, i.e., insertions are less frequent and updates/deletions are seldom necessary, content-based image retrieval systems in practice require a dynamic information storage structure.

IV. CIR-TREE

1. Characteristics

Various index structures for high dimensional data sets have been proposed. However most of them have a dimensionality problem, as surveyed in the previous sections. TV-tree and X-tree are representative index structures that have been introduced recently to support queries for high-dimensional data. It is true that they are more adequate index structures for high-dimensional data than existing ones such as R-tree and its variants. However, they still have some problems in dealing with overlaps and a large number of features as pointed out in Section II.

We propose a new high dimensional index structure, called CIR (Content-based Image Retrieval)-tree, in order to alleviate these problems. The proposed CIR-tree mostly satisfies the design requirements mentioned in Section III. The main idea of CIR-tree comes from the insights of the two structures, that is, the main characteristics of the X-tree and the TV-tree. We applied the main idea of both tree structures to CIR-tree in order to solve the dimensionality problem, and enhance the reinsert algorithm. For the nodes that are close to the root node, we use only a few dimensions so that we can store more branches and obtain a high fanout like the TV-tree. On the other hand, we used more and more dimensions as descending tree so that we can see more discrimination. In the CIR-tree, it is assumed that feature vectors for data objects are ordered in ascending order by its importance, and the importance can be obtained by employing various conversion functions [13].

Like other index structures, CIR-tree represents data with a hierarchical structure. A node in one level has its children nodes. This constitutes a hierarchical structure starting from a root node to leaf nodes. An internal node includes the MBRs of its children nodes, and a leaf node has feature vectors. The CIR-tree alleviates the disadvantages of the index structures of R-tree group. According to experimental evaluation of overlaps in the R*-tree directories, overlaps increase up to about 90% for high dimensionality larger than 5 [14]. Increased overlaps deteriorate the performance of an index structure remarkably. Overlaps can be increased when a node is split or a record is inserted. The CIR-tree employs the concept of supernodes in X-tree to avoid overlaps and uses a new split algorithm to minimize overlaps when an overflow occurs in a node. That is, the CIR-tree avoids or minimizes overlaps using the split algorithm whenever it is possible without degenerating the tree. Otherwise, the CIR-tree uses extended variable size directory nodes, so-called supernodes. Therefore, the structure of the CIR-tree is the mixture of the linear array structure for representing supernodes and the R-tree like hierarchical structure.

The CIR-tree achieves dynamic reorganizations by forcing entries to be reinserted during the insert operations, which regroups entries and thus decreases overlaps. The CIR-tree uses the concept of weighted center, or the average coordinates of each entry, to enhance the performance of the reinsert algorithm: using the weighted center significantly improves clustering effects of nodes in the CIR-tree. Consequently the CIR-tree constructs a condensed tree structure and also decreases the overlap between neighboring nodes.

In general, the existing index structures use Euclidean distance as a similarity measure on retrieval. However, the Euclidean distance may not be appropriate as a distance measure for high dimensional data because of its exactness limit. To alleviate such a problem, CIR-tree uses the weighted Euclidean distance such as in (1). The weighted Euclidean distance processes various kinds of similarity queries more efficiently than the Euclidean distance.

$$D(X,Y) = \sqrt{(x-y)^T \, diag(w)(x-y)}, \tag{1}$$

where, x and y are feature vectors and w is a vector representing a relative weight.

2. Structure of the CIR-tree

The structure of the CIR-tree is similar to that of TV-tree except for supernodes. Each node, except leaf nodes, consists of a set of entries, where each entry is composed of a pointer to a child node and an MBR of the node. The MBR is a minimum bounding region containing all entries of a node; it also has a feature vector that is agreed on the node's descendents in the

sense that the first, say, k dimensions of their feature vectors are inactive ones. Therefore, our CIR-tree provides a higher fanout at the top levels of an index structure.

The data structures for an MBR are as follows:

```
struct    MBR  {  Feature   inactive,
                  Feature   lower,
                  Feature   upper};
struct Feature {  float     feature_value[];
                  int   no_of_dimensions };
```

where `Feature` denotes 'feature vector.'

A directory node contains MBRs that represent minimum bounding regions of all their descendents. The data structure is as follows.

```
struct Branch_node   {   int  no_of_element;
                         list  of(MBR) };
```

A leaf node includes actual feature vectors. The structure of the leaf node is as follows.

```
struct Leaf_node     {   int  no_of_element;
                         list  of(Feature) };
```

A supernode is created when splitting a directory node. We will discuss the conditions of creating supernodes in Section IV-3.B, when we describe the split algorithm. The structure of supernodes is represented as a contiguous array of nodes.

3. Algorithms in CIR-tree

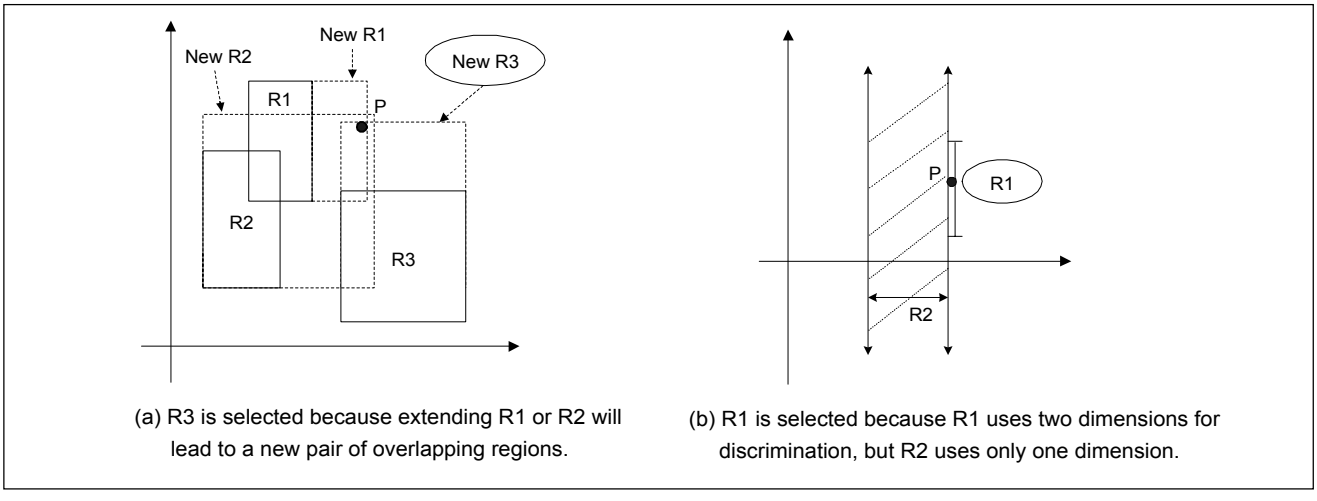
A. Insertion algorithm

To insert a new object, we should find the branch at each level that is most suitable to hold the new object, and then insert the new object to the chosen leaf node. If an overflow occurs at this time, we can cope with it by reinserting some entries in the node or splitting the node. After inserting, splitting, or reinserting a node, we update the MBRs of affected nodes.

The insertion algorithm calls ChooseSubtree algorithm first. ChooseSubtree algorithm chooses the most appropriate node using predefined criteria to insert a new object. The algorithm is important to make a well-clustered tree structure, and then to reduce overlaps and significantly improve retrieval performance. The algorithm ChooseSubtree uses the following criteria, in a descending priority:

① Select the MBR that will result in a minimum number of newly created overlapping MBRs within a node. Figure 3 (a) shows an example of inserting a new object with this criterion.
② Select the MBR that uses more dimensions for discrimination, that is, the MBR of which more dimensions are inactive. Figure 3 (b) shows only the first two dimensions. R1 and R2 are overlapped. R1 uses two dimensions for discrimination and R2 uses one dimension for it. R1 may have more regions in the direction of the next higher dimensions. When inserting the point P, R1 is selected rather than R2, because R1 uses more dimensions than R2. Using more dimensions means that similar objects are clustered in a small region.
③ Select the MBR whose center will be close to a new object.

When an overflow occurs during an insertion operation, the CIR-tree firstly tries to reinsert some of the entries in the node. If the CIR-tree cannot reinsert the entries, however, because the overflow has occurred while reinserting some other entries caused by another overflow, then it splits the node instead of reinserting the entries. If the area of overlaps within the node exceeds a certain threshold value in the split algorithm, the node is extended to a supernode. The detailed split algorithm will be explained in Section IV-3.C. The pseudo code for the insertion algorithm is as follows.



(a) R3 is selected because extending R1 or R2 will lead to a new pair of overlapping regions.

(b) R1 is selected because R1 uses two dimensions for discrimination, but R2 uses only one dimension.

Fig. 3. Illustration of the criteria of ChooseSubtree algorithm.

Algorithm Insertion

```
1. ChooseSubtree()  // choose the best branch
                    // to follow, descend the
                    // tree until the leaf
                    //  node is reached
2. Insert a new object into the leaf node.
3. if(node overflows)
4.     Call Reinsert
5.     if(Reinsert fail)
6.         Call Split
5.         if(the split routine returns supernode)
6.             Extend the leaf node to supernode
7.         else
8.             Insert the MBRs of the two split
               nodes into the parent node
9. UpdateTree()     // update the MBRs of the
                    // parent node
```

B. Reinsert algorithm

In most tree structures for high dimensional data, including R-tree, R*-tree and TV-tree, different sequences of insertions will result in different tree structures. MBRs that have been introduced during the early growth of the tree structure in some sequences may lead to a bad retrieval performance in the current situation. Therefore, we need a mechanism to reorganize the tree dynamically to avoid performance degradation.

The R-tree, the R*-tree and the TV-tree force the entries to be reinserted during the insertion routine to reorganize tree structures. However, the X-tree does not perform reinsertion, and thus it can have more overlapped bounding rectangles than other index structures. Supernodes are used in the X-tree to avoid overlaps; however, the size of supernodes may increase and then deteriorate the retrieval performance.

If an overflow occurs in a node, p entries farthest from the center of the node are deleted and then reinserted into the tree. This provides a possibility of eliminating dissimilar entries from the node so that it accomplishes more efficient clustering. The parameter p can be varied in performance tuning stage. The experimentally suggested value of p in [16] is 30% of the maximum number of entries in a node.

The R*-tree and TV-tree use geometric center to find the farthest entries. The CIR-tree uses weighted center. The weighted center $\vec{c}$ of a node N is defined as

$$\vec{c} = \frac{\sum_{i=1}^{n} \vec{e}_i}{n},$$

where $\vec{c}_i$ denotes the center vector of the entry and n denotes the number of entries in a node N.

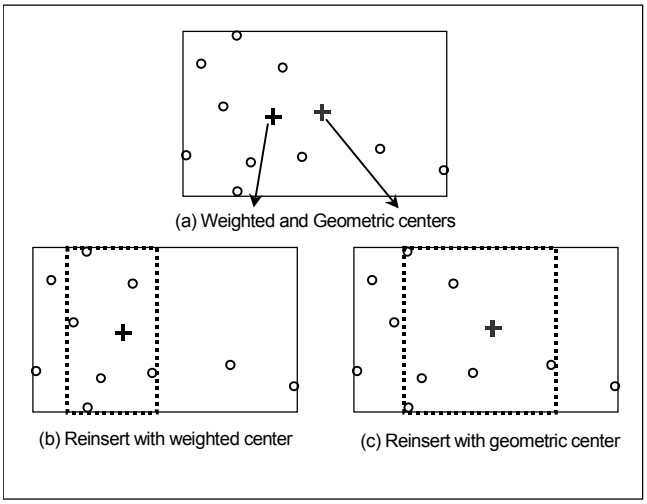


Fig. 4. Comparison of a weighted center with a geometric center where p = 40%. Reinsertion with weighted center makes smaller bounding rectangle.

If we use the weighted center as one of a node, the center of a node is moved toward a place where, in its vicinity, the entries are denser. Figure 4 shows the effect of the weighted center when $p = 40\%$. By using the weighted center, we can get smaller, or well-clustered MBRs, after deleting the farthest entries. In addition, the smaller MBRs would decrease overlaps. The computation cost for a weighted center will be dependent on the number of entries and the number of dimensions of each entry; however, it is not a great burden in the whole algorithm.

Algorithm Reinsert

```
1. delete p% of entries from a node
2. insert them into the tree
3. if overflow occurs during insertion
4.     return fail
5. else
6.     return success
```

C. Split algorithm

The purpose of splitting a node is to divide the set of MBRs into two groups in order to facilitate upcoming operations and provide high space utilization. The creation or extension of a supernode occurs if there is no possibility to find a suitable hierarchical structure. In other words, if dividing of the MBRs results in a large overlap split, we do not split the node but create a supernode of twice the size of the node, or append a block if the current node is a supernode.

When splitting a node, we sort the entries of the node by the first active dimension, then look for the best break point in the sorted entries where the overlap of the two split MBRs becomes the minimum. Of course, both of the two split nodes have a

larger size than the minimum fill factor. If the overlap exceeds *MaxO*, mentioned in Section II, the directory node would be extended to a supernode. In CIR-tree, we set the value of *MaxO* as 20%.

Algorithm Split

```
1. find the 1st dimension with which overlap
   free split is possible.
2. if the dimension found
3.      do split
4.      return the MBRs of the two nodes
5. else    // overlap-free split is impossible
6.      split with the first active dimension
7.      If(overlap_ratio > MaxO) return supernode
8.      else return the MBRs of the two nodes
9. end
```

D. Search algorithm

In this algorithm, a search starts from the root node. It examines each entry that intersects the search area, recursively following the entries, because MBRs are allowed to be overlapped. The following is the pseudo-code of the search algorithm:

Algorithm Search

```
1. If(accessed node == Leaf node)
2.    Evaluate the similarity of the query and
      the entries in the node.
3.    Return the objects satisfying the query
      according to similarity.
4. else // for directory nodes
5.    Select all MBRs including the query for
      active dimensions.
6.    Call the search algorithm recursively
      with each child node selected in step 5.
7. end
```

E. Nearest Neighbor Search algorithm

We used Hjaltason and Samet's algorithm known as optimal [23], [24]. This algorithm uses *MINDIST* as a metric to prune nodes from a search list; *MINDIST* is an Euclidean distance between the query point and the nearest edge of a rectangle. Since the CIR-tree and the TV-tree do not use full dimensions to compute *MINDIST*, the effectiveness of pruning can be degraded. However, the small number of directory nodes of CIR-tree compensates the ineffectiveness in pruning. The following is the pseudo-code of the nearest neighbor search algorithm.

Algorithm NN_Search

```
SearchQueue : Priority Queue
1. push the child nodes of the root and their
   MINDISTs into SearchQueue
2. while (SearchQueue is not empty)
3.    if(top of SearchQueue is a leaf)
4.       find nearest point NNP
5.       if(NNP is closer than NN)
6.            prune SearchQueue with NNP
7.            let NNP be the new NN
8.    else
9.       push its child nodes into Search-
         Queue
10       endif
11.   endwhile
```

F. Deletion

Deletion is quite simple unless underflow occurs. In this case, the remaining entries of the node will be deleted and reinserted. The underflow may be propagated to the upper levels of the tree.

V. EXPERIMENTS

We show the characteristics of the proposed CIR-tree by comparing its performance with those of TV-tree, X-tree and Pyramid-Technique. In the experiments, we used SUN SPARC station 20 with 128 Megabytes of main memory and 6 Gigabytes of hard disk. All simulation programs were implemented with ANSI C++ and compiled with a GNU C++ compiler. We used the TV-tree, the X-tree and the Pyramid-Technique programs without modifying the program sources that were implemented by the authors of [13], [14], [19]. The size of a standard block in these experiments is 4 KB. As a synthetic data set, we generated 2,000,000 uniformly distributed floating-point numbers between 0.0 and 10.0, and then we grouped them with desired dimensions to make data points. The dimension, say *D*, was varied from four with 500,000 data points up to 18 with 111,111 data points.

1. Retrieval Performance

Figures 5 and 6 show the performance of three index structures for exact match queries. We have applied 50,000 exact match queries to uniformly distributed data varying the number of dimensions of the data points, *i.e.*, each graph of the figures represents the average of 50,000 independent experiments. Note that, in order to count the number of page accesses, the access to a supernode of size *s* was counted as *s* page accesses. As shown in Fig. 5, the CIR-tree outperforms other index trees. Since the number of data points is decreased with growing dimensionalities of the data points, the number of page accesses of the TV-tree is reduced. The retrieval performance of the X-

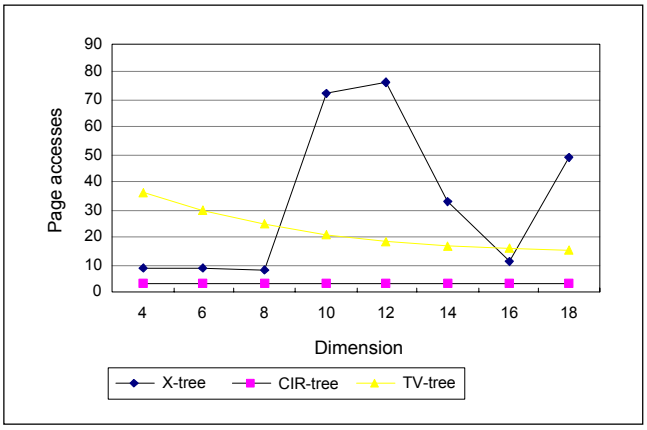


Fig. 5. The number of page accesses for an exact match query.

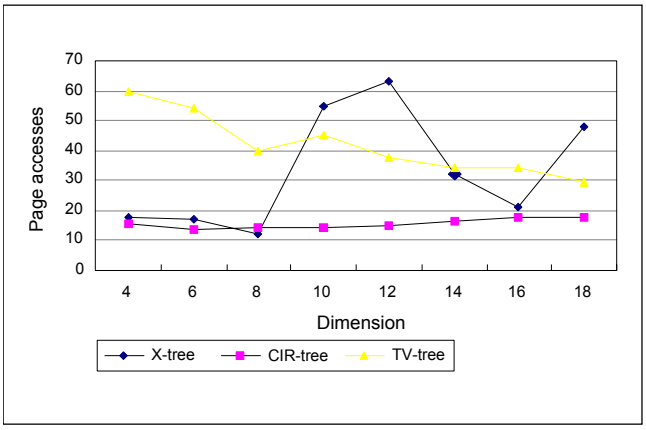


Fig. 7. Range query.

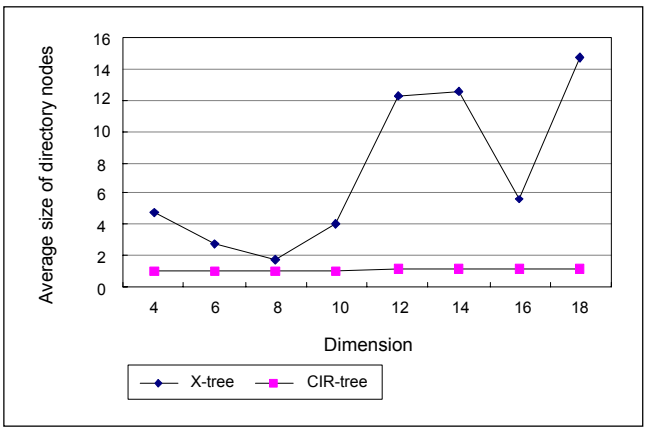


Fig. 6. An average size of directory nodes.

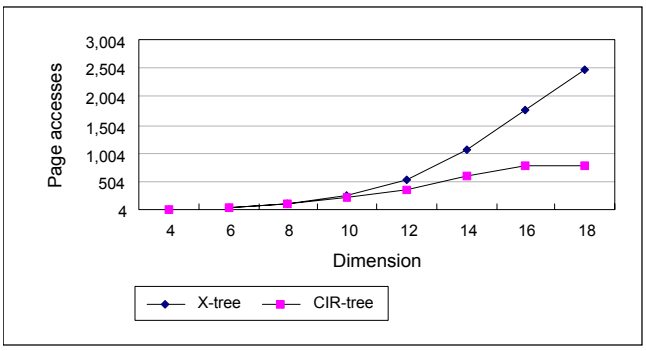


Fig. 8. Ten nearest neighbor query.

tree depends on the size of the supernodes. For example, when $D = 12$, the number of page accesses increases significantly, because the X-tree does not employ the forced reinsertion technique and the number and size of supernodes increase as the number of dimensions increases. The facts that no forced reinsertion technique is used and that different sequences of insertions may result in different tree structures may fluctuate the graph for the X-tree. However, the number and size of supernodes of CIR-tree are smaller than those of the X-tree even though CIR-tree uses supernodes, because the CIR-tree maintains smaller directory sizes. Eventually the CIR-tree always provides better performance for exact match queries over the other index structures.

We also performed range queries and nearest neighbor queries with the TV-tree, the X-tree and the CIR-tree. For the range queries we first generated 5,000 center vectors using random number generation, and then made two range-bounding rectangles: for the upper bound vector we added one to each dimension of the center vector; for the lower bound vector we subtract one from the center vector. We extended the TV-tree to process the range queries and the neighbor queries, because the original TV-tree supports point queries only. The results of the range queries are presented in Fig. 7. Similar to the results shown in Fig. 5, the number of page accesses of the X-tree depends on the size of supernodes and the number of page accesses of the CIR-tree seems stable.

Figure 8 shows the results of 10 nearest neighbor queries. We only compare the performance of the proposed CIR-tree with that of X-tree, because the performance of the k nearest neighbor queries with the TV-tree are so poor that it is meaningless to present the results as a graph in the figure. The number of page accesses of the X-tree increases exponentially with dimension. On the other hand, the number of page accesses of the CIR-tree increases linearly with dimension.

Finally, we compared the CIR-tree with the Pyramid-Technique by using a real data set. We used the letter image recognition data in [25] as the real data set, which consists of 20,000 points of 17-dimensional data (1 category and 16 numeric features). The category is one of the 26 capital letters in the English alphabet and the numeric features are scaled to fit into a range of integer values from 0 to 15. Figure 9 shows the experimental results with the real data. In this figure we can see that the CIR-tree provides better performance than the Pyramid-Technique and X-tree in range queries and nearest neighbor queries. Note that, in Fig. 9

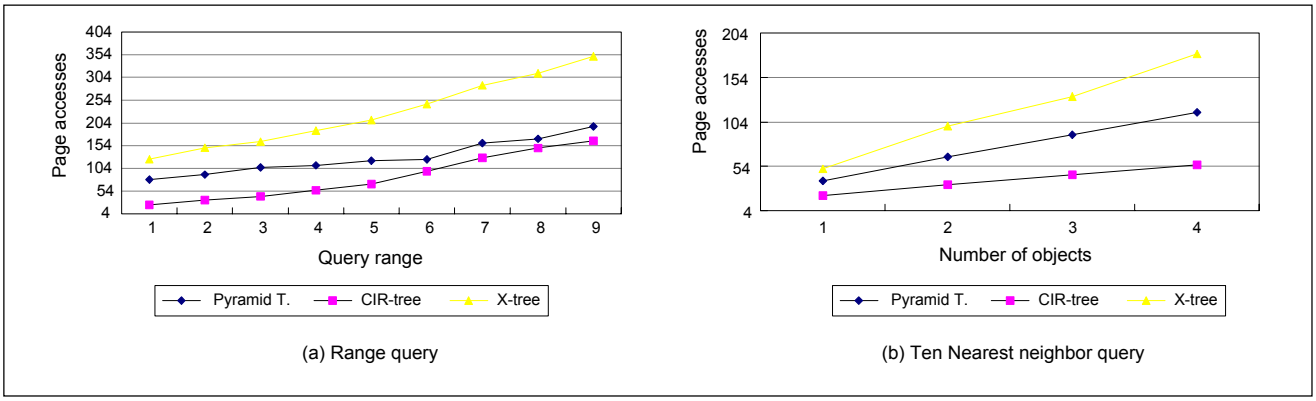


Fig. 9. Comparison with Pyramid Technique and X-tree by using the real data.

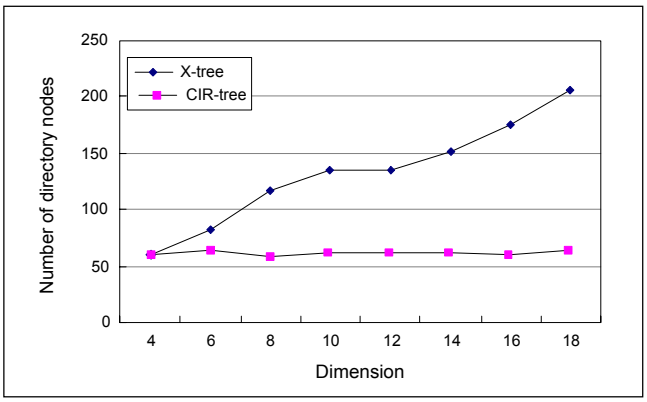


Fig. 10. Number of directory nodes depending on the dimensionality.

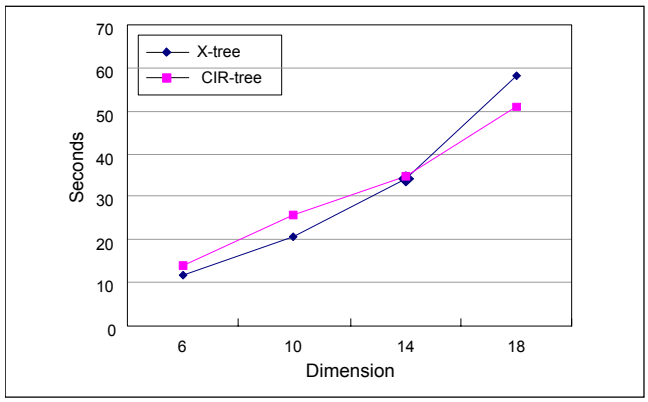


Fig. 11. Insertion Time of CIR-tree and X-tree.

(a), the difference of the number of the page accesses of them becomes smaller as the range increases. The performance of the CIR-tree seems better than those of the Pyramid-Technique and X-tree in large ranges, but we cannot definitely say that it is superior to the Pyramid-Technique because the Pyramid-Technique has a simpler node structure than the CIR-tree. In other words, although the Pyramid-Technique needs more page accesses than the CIR-tree, it may spend less CPU time than the CIR-tree because of its simple node structure.

2. Storage Space

Figure 10 shows an experimental result of each index structure in terms of storage space. Due to the fact that the CIR-tree and X-tree create similar number of leaf nodes, the comparison of the number of leaf nodes is meaningless. So we only compared the number of directory nodes. The figure shows that the space usage of the X-tree increases with dimension, but the space usage of the CIR-tree is stable. This is because the CIR-tree stores a small number of features in the directory node for all dimensions. The CIR-tree creates a small number of nodes and uses the storage space effectively. As a result, the performance comparison in terms of storage space shows that the storage overhead of the CIR-tree is much less than that of the X-tree.

3. Insertion Performance

Because the CIR-tree uses a reinsertion technique in the insertion algorithm, the cost for insertion operations is higher than those of the X-tree and the Pyramid-Technique which do not use reinsertion technique. The Pyramid-Technique always spends less time than CIR-tree since it uses B^+-tree, so its insertion time is B^+-tree construction time and transformation time. The X-tree spends less time than CIR-tree, but as the size of the supernode grows, the X-tree tends to spend more time than CIR-tree. Even though the CIR-tree performs reinsert operations, the number of directory nodes is much smaller than that of X-tree especially in a higher-dimension data space. Figure 11 shows the insertion time of X-tree and CIR-tree. The number of data objects was fixed as 10,000 and the dimension varied from 6 up to 18.

VI. CONCLUSIONS

In this paper, we have analyzed existing index structures for high-dimensional data and proposed several desired design re-

quirements that a new index structure must have. We have also proposed a new efficient index structure, called CIR-tree, which satisfies the design requirements. The proposed CIR-tree uses fewer dimensions at the upper levels, which allows an index structure to have a higher fanout at the top levels and a node to hold more data. The CIR-tree makes the height of a tree become lower, solving the dimensionality problems; it supports high dimensional data more efficiently, reduces the number of disk accesses, and improves the disk storage utilization. The CIR-tree produces well-clustered index structures by using the weighted center in the reinsert algorithm as well as using supernodes to avoid overlaps. The CIR-tree also uses a weighted Euclidean distance to overcome the exactness problem of Euclidean.

We have compared the proposed CIR-tree with the TV-tree, the X-tree and the Pyramid-Technique through various experiments to manifest the superiority of CIR-tree. The experiments show that the CIR-tree outperforms the TV-tree, the X-tree and the Pyramid-Technique in terms of retrieval speed and space requirements. But the CIR-tree needs further investigation to improve nearest neighbor query performance.

REFERENCES

[1] W. E. Mackay and G. Davenport., "Virtual video editing in interactive multimedia applications," *Communications of the ACM*, July 1989, Vol. 32, pp. 802–810.

[2] Myron Flickner and *et. al*., "Query by Image and Video Content : The QBIC System," *IEEE Computer*, Vol. 28, No. 9, 1995.

[3] Charles E. Jacobs, Adam Finkelstein, and David H. Salesin., "Fast Multiresolution Image Query," *Proc. of the 1995 ACM SIGGRAPH*, New York, 1995.

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

[5] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equiz, "Efficient and Effective Querying by Image Content," *Journal of Intelligent Information System (JIIS)*, Vol. 3, No. 3, July 1994, pp. 231–262.

[6] B. Furht, S. W. Smoliar, and H. Zhang, "Video and Image Processing in Multimedia Systems," *Kluwer Academic Publishers*, 1994.

[7] Y. Alp Aslandogan, Chuck Their, Clement T. Yu, Chengwen Liu, and Krishnakumar R. Nair, "Design, Implementation and Evaluation of SCORE (a System for Content based Retrieval of pictures)," *Proc. of 11th international conference of Data Engineering*, 1995, pp. 280–287.

[8] P. M. Kelly, T. M. Cannon and D. R. Hush, "Query by image example: the CANDID approach.," *Proc. SPIE Storage and Retrieval for Image and Video Database III*, Vol. 2420, 1995, pp. 238–248.

[9] J. K. Wu, A. Desai Narasimhalu, B. M. Mehtre, C. P. Lam, and Y. J. Gao, "CORE: a content-based retrieval engine for multimedia systems," *ACM Multimedia Systems*, Vol. 3, 1995, pp.25–41.

[10] M. J. Swain and D. H. Ballard., "Color indexing," *International Journal of Computer vision*, Vol. 7, No. 1, 1991, pp. 11–32.

[11] D. A. White and R. Jain, "Similarity Indexing with the SS-tree," In *Proc. 12th Intl. Conf. On Data Engineering,* New Orleans, 1996, pp. 516–523.

[12] D. A. White and R. Jain, "Similarity Indexing: Algorithms and Performance," In *Proc. of the SPIE : Storage and Retrieval for Image and Video Databases IV*, Vol. 2670, 1996, pp. 62–75.

[13] K.I. Lin, H. Jagadish, and C. Faloutsos, "The TV-tree: An Index Structure for High Dimensional Data," *VLDB Journal*, Vol. 3, 1994, pp. 517–542.

[14] S. Berchtold, D. A. Keim, and H-P. Kriegel, "The X-tree: An Index Structure for High-Dimensional Data," *Proc. of the 22nd VLDB Conference*, Bombay, India, 1996.

[15] Norio Katayama, and Shin'ichi Satoh, "The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries," *Proc. ACM SIGMOD Int. Conf. On Management of Data*, Tucson, Arizona, 1997, pp. 369–380.

[16] N. Beckmann, H.P. Kriegel, R. Schneider and B. Seeger, "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles," *ACM SIGMOD*, May 1990, pp. 322–331.

[17] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest Neighbor Queries," *Proc. ACM SIGMOD Int. Conf. On Management of Data*, San Jose, CA, 1995, pp. 71–79.

[18] Roger Weber, Hans-Jorg Schek, and Stephen Blott, "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces," *Proc. of the 24th VLDB Conf.*, New York, USA, 1998, pp 194–205.

[19] Stefan Berchtold, Christian Bohm, and Hans-Peter Kriegel, "The Pyramid-Technique: Towards Breaking the Curse of Dimensionality," *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Seattle, WA, 1998, pp. 142–153.

[20] A.Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching," *Proc. 7th Int. Conf. on Data Engineering*, 1991, pp. 520–527.

[21] D. Comer, "The Ubiquitous B-tree," *ACM Computing Surveys*, Vol. 11, No. 2, 1979, pp.121–138.

[22] Y. Gong *et al*., "An Image Database System with Content Capturing and Fast Image Indexing Abilities," In *Proc. of the International Conf. on Multimedia Computing and Systems*, IEEE, Boston, MA, May 1994, pp. 121–130.

[23] Gisli R. Hjaltason and Hanan Samet, "Ranking in spatial Databases," *Proc. of the 4th Symposium on Spatial Databases*, Portland, Maine, Aug. 1995, pp. 83–95.

[24] Andreas Henrich, "A Distance Scan Algorithm for Spatial Access Structures," *ACM-GIS*, 1994, pp. 136–143.

[25] C. L. Blake and C. J. Merz, "UCI Repository of Machine Learning Databases," Irvine, University of California, Department of Information and Computer Science, 1998.

Jang Sun Lee received the B.S. degree in Computer engineering from Kyungpook National University, Taegu, Korea in 1983, the M.S. degree in computer science form Korea Advanced Institute of Science and Technology (KAIST), Taejon, Korea in 1985, and his Ph.D. in computer science from Syracuse University in 1997. He joined ETRI in 1985 and has worked for the development of system software technologies. Currently he is the head of Storage System Software Research Team. His current research interests include parallel I/O, SAN and NAS, database systems, parallel and distributed systems, and operating systems.

Jae Soo Yoo received the B.S. degree in Computer engineering from Chonbuk National University in 1989, and the M.S. and Ph. D. degrees in computer science form Korea Advanced Institute of Science and Technology (KAIST), in 1991 and 1995, respectively. From March 1995 to August 1996, he was a faculty member of the department of computer science at Mokpo National University. He has been an assistant professor in the department of computer and communication engineering at Chungbuk National University since 1996. His research interests include database systems, multimedia database, information retrieval and distributed object computing

Seok Hee Lee received the B.S. and M.S. degree in the department of computer & communication engineering from Chungbuk National University, Cheongju Korea, in 1994 and 1998, respectively. Since 1998, he has been studying database systems at the department of computer & communication engineering in Chungbuk National University for his Ph.D. Since March 2000 he has been with the department of internet broadcasting at the Dong-Ah Broadcasting College, Anseong Korea, where he is currently a full-time lecturer. His research interests include multimedia and image database systems, information retrieval, image processing and image communication.

Myung-Joon Kim received the B.S. degree from Seoul National University, Korea, the M.S. degree from KAIST (Korea Advanced Institute of Science and Technology) Taejon Korea and the Ph.D. degree from University of Nancy I, Nancy France in 1978, 1980 and 1986 respectively, all in computer science. He joined ETRI in 1986 and has worked for the development of system software technologies especially database systems and distributed system technologies. He served as head of Database Section (1989-1992), head of Software Engineering Section (1994), director of System Software Department (1995-1997), director of Database Technology Department (1998) and director of Internet Service Department (1999). In 1993 he worked at University of Nice Sophia-Antipolis, France as a visiting professor. Currently he is Vice President of Computer & Software Technology Laboratory. His current research interests include database system, transaction processing, distributed systems, real-time OS, object technologies and their deployment for new Internet applications.