An Indexing and Retrieval Mechanism for Complex Similarity Queries in Image Databases

GUANG-HO CHA ghcha@tmic.tit.ac.kr
Department of Multimedia Engineering, Tongmyong University of Information Technology
Pusan 608-711, South Korea

CHIN-WAN CHUNG chungcw@islab.kaist.ac.kr
Department of Computer Science, Korea Advanced Institute of Science and Technology
Taejon 305-701, South Korea

Abstract. A content-based image retrieval mechanism to support complex similarity queries is presented. The image content is defined by three kinds of features: quantifiable features describing the visual information, non-quantifiable features describing the semantic information, and keywords describing more abstract semantic information. In correspondence with these feature sets, we construct three types of indexes: visual indexes, semantic indexes, and keyword indexes. Index structures are elaborated to provide effective and efficient retrieval of images based on their contents. The underlying index structure used for all indexes is the HG-tree [1]. In addition to the HG-tree, the signature file and hashing technique are also employed to index keywords and semantic features. The proposed indexing scheme combines and extends the HG-tree, the signature file and the hashing scheme to support complex similarity queries. We also propose a new evaluation strategy to process the complex similarity queries. Experiments have been carried out on large image collections to demonstrate the effectiveness of the proposed retrieval mechanism.

1. Introduction

Many research groups are developing the multimedia information systems that will perform the storage, browsing, indexing, and retrieval of multimedia data. The representative systems are QBIC [2], Chabot [3], CAFIIR [4], and STAR [5] and so on. These systems are designed to allow users to retrieve images based on their contents. They have focused on the convenient user interface, querying methods, and system performance problem. However, they have not addressed the problems of modeling of the image contents, indexing of the image contents and processing of complex queries *as a whole*.

This paper describes the design and implementation of the content-based image retrieval system. This research has three goals: (1) to develop the model to describe the image content, (2) to develop the index structures to speed up the search, and (3) to investigate the processing techniques for complex queries.

Our first problem is to characterize the contents which describe the image. The contents to describe the image are extracted automatically from the image by image processing subsystem or annotated manually by human intermediaries. The selection of the contents influences largely on the retrieval effectiveness of the system. The more the image include content information, the better the retrieval system can discriminate the requested ones among the images in the database. As more images are inserted into the database, more content information might be

needed to discriminate the desired ones. Also, the determination of the content of image is influenced by application domain. For example, some applications or users require visual information to retrieve images (e.g., “Find images that have approximately 30% red and 15% blue colors) and some other applications need semantic (i.e., not quantifiable but perceptible by human) information (e.g., “Find images with beautiful mountain scenery”). As more information about the image contents are specified in the query predicates, more accurate query results might be acquired. To sum up, the content description of an image should comprise many useful information to support a wide range of queries effectively.

The second problem addressed is the development of efficient indexing scheme that makes use of the characterized contents of image. We need separate indexes for the feature sets describing the image content, because the features have their own intrinsic characteristics which cannot be mixed together. For example, the visual features such as colors and textures inherent in the image are quantifiable, on the other hand, the semantic features or keywords obtained only by human sense cannot be quantified by any metric function.

Indexing tabular data for exact-match search or range search in traditional databases is a well-understood problem, and the index structures like B-tree family [6] provide efficient access mechanisms. However, they are not likely to provide enough information to deal with complex image contents. The one-dimensional B-tree does not usually reflect the *n*-dimensional domain space, where *n* is the number of image features to index, and thus the B-tree representation is not particularly conducive to *n*-dimensional queries. In addition, B-trees may not be appropriate to the similarity searching for multimedia content. For queries in which similarity is defined as a distance metric in high-dimensional feature spaces, the indexing involves clustering of the objects in the multidimensional space and indexable representations of the clusters. Therefore, the traditional index structures such as B-trees are not appropriate for image data. Index structures to provide fast accesses in high-dimensional feature space must be provided. Moreover, the index structures indexing the narrative text should also be investigated according to the content model of the image.

The third problem is to develop the processing techniques for complex queries consisting of more than one similarity predicates, which are called *complex similarity queries* [7]. Efficient processing of queries using various features is an essential requirement in content-based image retrieval systems. The processing of complex similarity queries require to have some efficient evaluation strategy. The problem comes from the fact that the types of query results to individual predicates are different. For example, the type of query result to one predicate is a set of values, on the other hand, the type of query result to the other predicate is an ordered list. Some

approaches to solve to this problem can be found in [7, 8].

With these three issues to be solved for constructing multimedia information system, we have developed the prototype content-based image retrieval system which is able to support a wide range of queries. The rest of the paper describes the design of our system. In Section 2, our content model which describes the representation of the image is presented, and the types of queries that are dealt with in our prototype system are also provided. In Section 3, we establish three index structures for three types of image features. Section 4 describes the processing method for complex similarity queries. In Section 5, the overall indexing and retrieval mechanism for complex similarity queries are described. In Section 6, the architecture of our image retrieval system and the experimental results of query processing are provided. The conclusions are made in Section 7.

2. Images and Queries

For images, during database population (i.e., the process of creating a database) images are processed to extract features describing their content, and the extracted features are stored as *metadata* in an image database. The effectiveness of content-based retrieval depends largely on the availability of rich metadata. Using these metadata, indexes are constructed and queries are specified. In this section, the model to describe the image content is presented and the queries to be performed on this image model are described.

2.1 Description of an Image

The basic system unit that may be stored and retrieved is an *image object*. An *image object* I consists of a *body* B and *a header* H. The body is a binary bitmap which has a specific image format such as JPEG, GIF, BMP, and so on. The header is a metadata that describes the content of the image. We model the header as a triple $H = (A_v, A_s, A_k)$, where

- A_v is a set of *visual feature values*,
- A_s is a set of *semantic feature values*,
- A_k is a set of *keywords*.

A_v consists of fixed number of visual features which may be extracted automatically by the image processing subsystem. The visual features would be colors, textures, shapes, positions, and so on. A_s consists of fixed number of semantic (i.e., non-quantifiable or abstract) features that should be extracted manually by the interpretation of human intermediaries. The semantic features contain higher level of abstract information than the visual features. For example we can

enumerate the following semantic features of an image:

- title: title of image,
- subject: peace, love, sea, animal, flower, architecture, and so on,
- type: painting, scenery, portrait, and so on,
- perspective: aerial, ground, or close-up,
- orientation: horizontal or vertical,
- date: date when the picture is shot.

Some of these semantic attributes can be deduced by the system at the time the image is inserted into the database.

A_k consists of variable number of keywords. Keywords give the gist of an image. They are words or sequence of words which describe the characteristics of the image that can not be easily represented with only common semantic features. Keywords contain the highest level of content abstraction among other features. The maximum number of keywords allowed per image is a system parameter. Figure 1 shows an example header information that can be inserted into an image database.

A_v	*visual features*	dominant colors, textures, and object shapes
A_s	*title*	Waterloo bridge
	type	scenery
	subject	bridge
	perspective	aerial
	orientation	horizontal
	date	11-25-97
A_k	*keywords*	travel, ship, bridge, sunny

Figure 1. Sample image header

2.2 Description of a Query

The power of content-based retrieval in multimedia information systems must be increased if various types of queries from simple queries to complex queries can be supported. A *user query* is the specification of a header that closely corresponds to the information known about the image. Users can query the image databases based on the visual features, semantic features, and keywords. Some feature values may be omitted, or may be given a specific values or range of values. Keywords are specified by providing a list of words that describe the image. Visual

features are given by an example image, user-sketched drawings, or selected color and texture patterns.

To summarize the different types of queries, we have the following:

- *Simple queries* that specify a single attribute value for each possible attribute and require the exact match;
- *Range queries* that either explicitly specify a range of values for some of the attributes, as in (10% ≤ red ≤ 30%) and (30% ≤ green ≤ 40%) and (80% ≤ blue ≤ 90%), or implicitly specify a range of values by leaving one or more attributes values unspecified.
- *Similarity or nearest neighbor queries* that give an example image or user-sketched drawing and require to find most similar images to a given image.
- *Complex similarity queries* that consist of one or more similarity queries and other types of queries. For example, retrieve 5 images which are most similar to a given image and whose subject is '*animal*'.

The result of simple queries and range queries is a set of objects that satisfy the query predicates. On the other hand, the result of similarity or nearest neighbor queries is a list of objects ordered by their similarities. Since the complex similarity query has more than one type of queries including the similarity query, the individual query result may have different types of result sets. For example, if the query is composed of two subqueries (e.g., a similarity query and a range query), then two partial query processing results have two different types of results, i.e., a sorted list of objects and a set of qualified objects. The complex similarity query processing must synthesize partial two query results in a consistent manner. Since the complex similarity query covers all other types of queries, we will focus our attention on the complex similarity query in Section 4. The other types of query processing methods can be found in [1]

3. Indexing Scheme for Complex Similarity Queries

This section describes the indexing scheme that deals with the complex similarity queries. For a small database, sequential scanning of the features followed by straightforward similarity computations is adequate. But as the database grows, this procedure can be too slow, and indexes should be built to speed up the queries. Our indexing scheme consists of three types of indexes which correspond to visual features, semantic features, and keywords of image objects in image database.

3.1 Visual Indexes

We assume that a set of *n* visual features have been extracted automatically or manually from each image. They may be dominant colors, textures, shapes, and so on. When we represent a set of *n* visual features as an *n*-sized tuple, $A_v = (f_1, f_2, ..., f_n)$, it can be mapped to a point in an *n*-dimensional visual feature space. We use the HG-tree [1] as our underlying index structure for organizing the visual feature based indexes. The HG-tree is a multidimensional point index structure to index point data in multidimensional domain space. The HG-tree has outperformed most of other multidimensional point index structures in a wide range of query performance comparisons [1]. It has guaranteed storage utilization of 66.7% in worst case, and it has typically achieved more than 80%. It also has been shown that the HG-tree is fairly robust with respect to high dimensionalities because its *fanout* (i.e., the number of entries an index node can hold) is rarely influenced by the increase of dimensionality.

We could use other index structures for visual feature based indexes. In fact, many other image retrieval systems have used some other index structures. The QBIC system adopted the R*-tree [9] as an index structure. Petrakis and Faloutsos [10] used R-tree [11]. Mehrotra and Gary [12] used the K-D-B-tree [13]. The systems, CAFIIR [4] and STAR [5] and Zhang and Zhong [14] employed the iconic index tree based on the Self-Organizing Map (SOM) [15]. Each of these methods has not only its own advantages, but also has some limitations.

Compared with the other index structures, the performance of the multidimensional spatial index structures such as R-tree and R*-tree degenerates drastically with an increase in the dimensionality of the underlying feature space, because their fanout decreases in inversely proportional to the dimensionality. All current spatial index structures suffer from this *dimensionality curse* (i.e., exploding exponentially with the dimensionality). The improvement of the fanout is very important for the performance of the index structure.

The iconic index trees based on the SOM simplify the multidimensional problem by converting it to a one-dimensional clustering problem based on similarities. The major problem of these kind of index structures is that they are *static*. Usually the iconic index is constructed against a large data set which can represent the statistics of the data. However, if the system is to include other classes of data that change the node characteristics, then the system must be trained again in order to provide effective clustering. Therefore, an update due to insertion and deletion is accumulated and actually performed when the amount of update is up to a threshold. Another problem of the iconic index trees is that they are constructed only for nearest neighbor queries. Thus, it is difficult to process range queries. In fact, most of the index structures designed only for nearest neighbor queries have these common problems. For example, the optimistic VP (vantage point)-tree [16] and the GNAT (Geometric Near-neighbor Access Tree)

[17] are such kind of index structures. They precalculate some nearest neighbors of points, store the distances in a tree or graph, and use the precalculated information for a more efficient nearest neighbor search. Therefore, they have benefit in the nearest neighbor search time, but have disadvantage in update operations (insert and delete). In other words, they are static methods. Like the iconic index structure, another problem of these methods is that it is also difficult to process the range queries.

The HG-tree, which is one of multidimensional point index structures, avoid all above problems. It is less influenced than the spatial index structures by the increase of the dimensionality, because it represents each directory region covered by the data set by using only two Hilbert values regardless of the dimensionality. In addition, the HG-tree is completely dynamic, i.e., it supports arbitrary insertions and deletions of objects without any global reorganizations and without any loss of performance.

3.1.1 Basic Idea of the HG-tree

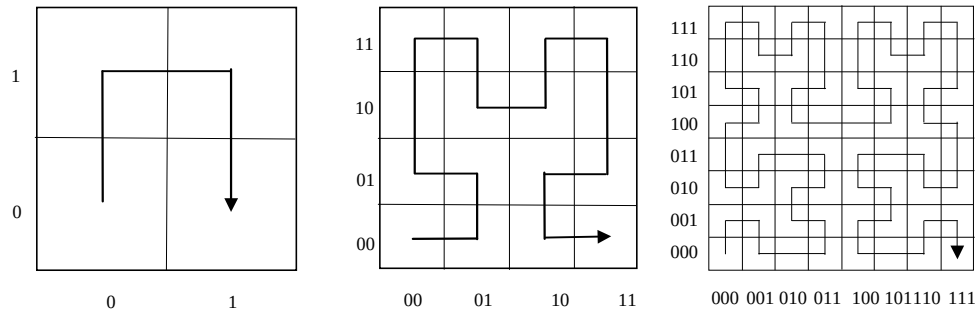
Physical storage of data objects ultimately requires a serialized ordering of the objects. To obtain this serialized order while retaining locality of neighboring objects in multidimensional data space, we map the objects with *n*-dimensional value to the 1-dimensional ones by using *Hilbert space-filling curve* [18]. A *space-filling curve* is a mapping that maps the unit interval onto the *n*-dimensional unit hyper-cube continuously. While there are other space-filling curves such as the Peano curve (also known as the Z curve) [19] and the Gray-code curve [20], it was shown that the Hilbert curve achieves better clustering than the others [21, 22]. The desirable features of the Hilbert curve are that the points close on the Hilbert curve are close in the domain space, and the points close in the domain space are likely to be close on the Hilbert curve.

The Hilbert curves of degree 1 (H_1), degree 2 (H_2), and degree 3(H_3) on a 2-dimensional data space are shown in Figure 2. The Hilbert curve can be generalized for any higher dimensionality.

In the HG-tree, all data points (e.g, a tuple A_v of visual features) are represented by locations on the Hilbert curve, and there is no need to consider the *n* dimensionality of the domain space. The reason for the transformation is to apply a linear ordering on the data points and on the index nodes.

Let us consider the structure of the HG-tree. The HG-tree consists of internal and leaf nodes. A *leaf node* contains at most C_l entries of the form

(oid, H)



(a) H_1 (b) H_2 (c) H_3

Figure 2. Hilbert curves of degree 1(H_1), 2(H_2), and 3(H_3)

where C_l is the capacity of the leaf node, *oid* is a pointer to the object in the database, and H is the Hilbert value for n-dimensional feature vector. An *internal node* contains at most C_n entries of the form

$$(ptr, I)$$

where C_n is the capacity of an internal node, *ptr* is a pointer to the child node, and I is the *minimum bounding interval* (MBI), which is the smallest interval on the Hilbert curve that covers all regions of the lower nodes. The MBI is represented by two Hilbert values at both ends of the interval:

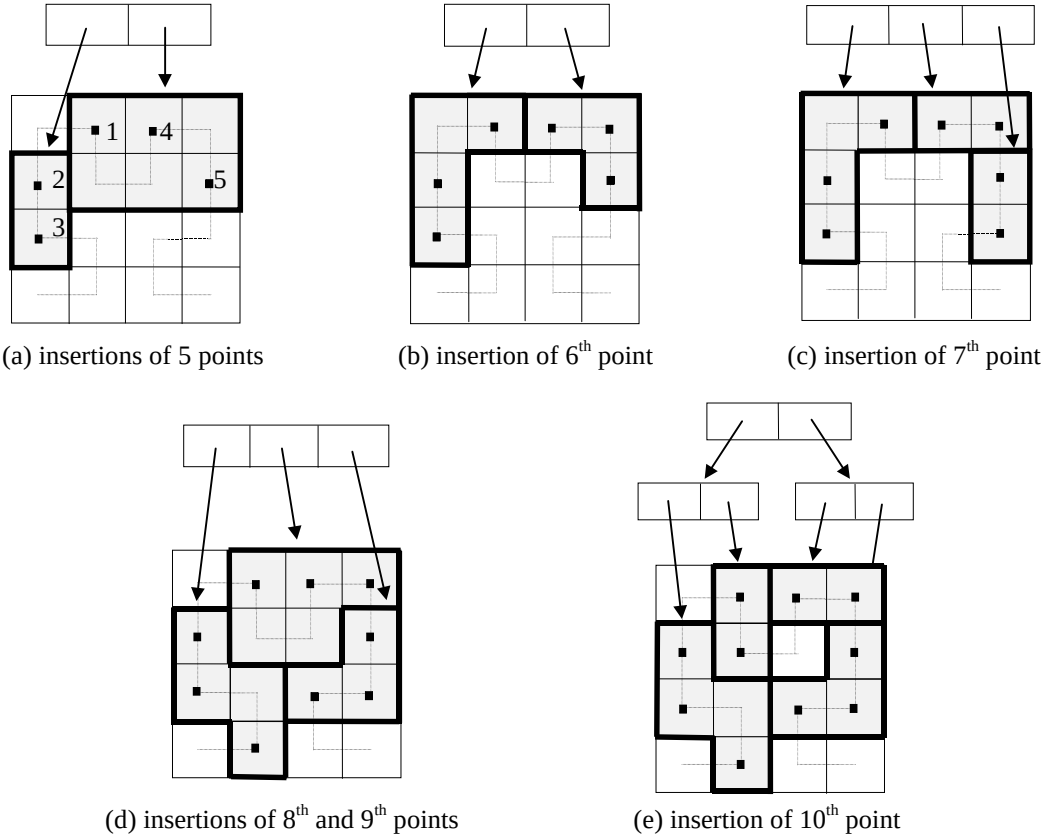
$$I = (H_1, H_2),$$

where H_1 is the starting point and H_2 is the ending point on the bounding Hilbert curve. We maintain these entries in Hilbert order. Using the MBI the HG-tree reduce the *dead space* (i.e., the space which does not include any actual data but covered by the directory region) of the node.

In the HG-tree, since all objects and directory nodes can be ordered through their Hilbert values, every node has a well-defined set of siblings. The HG-tree can absorb the node split caused by overflow by redistributing the objects of overflowing node into adjacent sibling and by adjusting their MBIs. When a split is unavoidable it converts the *two* nodes, i.e., the overflowing node and one of two adjacent siblings, into three. Resulting from this, the HG-tree guarantees that the worst-cast storage utilization is more than 66.7% (2/3) of full capacity. The experimental results show that the average storage utilization of the HG-tree is always more than 80%. With the example 1 we intend to visualize the basic properties of the HG-tree.

Example 1. Let the dimensionality be 2, the domain space be a 4×4 grid, and both of the fanouts of a directory page (= internal node) and a data page (= leaf node) be 3. The snapshots in Figure 3 depict the growth of the HG-tree. Each square and the number beside it represent a data point and the order of it inserted, respectively. In the data pages the actual points are stored.

The MBIs having at most 3 points are depicted by a fill pattern. The white area corresponds to empty space not managed by the HG-tree. When the 6*th* point is inserted, the data page is not split but points are distributed into its adjacent sibling and the directory regions are adjusted as in Figure 3(b). When the 7*th* point is inserted, two nodes are split into three as in Figure 3(c). The insertion of the 10*th* point makes an overflow of data page and in turn directory page increasing the height of HG-tree.



(a) insertions of 5 points (b) insertion of 6th point (c) insertion of 7th point

(d) insertions of 8th and 9th points (e) insertion of 10th point

Figure 3. The sequence of insertions of 10 points on the HG-tree

3.1.2 Shortcoming of the Hilbert mapping

Let us discuss the case when the Hilbert mapping causes negative impact on the performance of the HG-tree. Consider the worst case example as shown in Figure 4(a). Let us assume that the fanout of a node is 2. The circles denote data points. The size of MBI is pretty large when two data points lie on grid cells 1 and 14 as shown in Figure 4(a). However, even in that case, if two more points are inserted to the grid cells 2 and 13, then the node should be split into two and the new MBIs are constructed as shown in Figure 4(b). The size of the original MBI has greatly shrunk. When the size of database is large, this is the normal case we may assume,

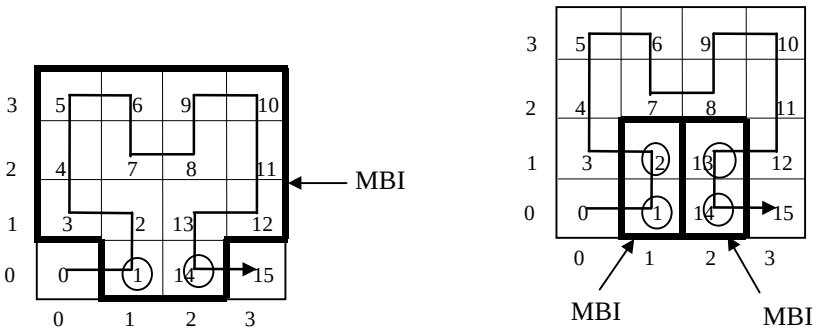


Figure 4. A worst-case example of the Hilbert mapping

the worst case as shown in Figure 4(a) rarely happens. Though the size of MBI may be large at first, it decreases gradually as the data objects are inserted as shown in Figure 4(b).

The reason why this kind of worst-case phenomenon happens is that while each grid point in *n*-dimensional grid has 2*n* nearest neighbors, the point in 1-dimensional line has only 2·1 nearest neighbors. Therefore, the mapping from *n* dimensions to 1 dimension has this problem inherently. However, in any case, the physical storage of data objects in *n*-dimensional space requires a linear ordering of the objects. Therefore, the best that one can hope to do in a mapping from *n* dimensions to 1 dimension is to have two of the 2*n* nearest neighbors to be nearest neighbors in the linear mapping. The Hilbert mapping satisfy this hope.

3.2 Semantic Indexes

A fixed-set of tuple A_s to describe the semantic features is represented by (s_1, s_2, … , s_k) such that $s_i \in D_i$ for $1 \le i \le k$. Where D_i for $1 \le i \le k$ is a domain space of the *i*-th semantic attribute s_i. We have *k* hash functions H_i: $D_i \rightarrow W_i$, for $1 \le i \le k$, where $W_i = \{0, 1, 2, ..., 2^{imax} - 1\}$, and $2^{imax} - 1$ is the maximum allowable hashed value. We construct the semantic indexes with this hashed tuple $H_s = (H_1(s_1), H_2(s_2), ..., H_k(s_k))$. Since this hashed tuple can be interpreted as a point in a *k*-dimensional domain space, we can use the HG-tree as in the visual indexes. Duplicate hashed tuple may be introduced due to collisions. If a data page is filled up by duplicate tuple, the HG-tree simply chain multiple data pages. However, such hash collisions for all organizing attributes will not be frequent.

Unlike other image/video retrieval systems such as QBIC and Chabot which use B-tree as their index structure to index semantic or tabular data, we use the HG-tree as our underlying index structure for semantic indexes because of its performance advantages. Since the B-tree is a primary index structure, the index is constructed only on a primary attribute. If it is required

to provide a fast access on other attributes, another B-trees constructed on that attributes are also needed. Instead of constructing multiple single-attribute (MSA) B-tree indexes, we construct a single multi-attribute (SMA) index, i.e., the HG-tree.

There are important advantages of using SMA index as compared to MSA index. First, the clustering of index pages and data pages on disk can drastically reduce the number of I/O operations needed for database accesses. Second, in the case of multiattribute queries, multiple independent accesses to separate indexes are performed and the intersection of the multiple partial results are needed to get a final result. This is very expensive functions. Third, when new records are inserted into or deleted from a database, SMA index organization needs only single update for its index. MSA index, in contrast, require multiple updates since the number of indexes is multiple. Therefore, maintaining the consistency of indexes in SMA index organization is simpler than that in MSA index organization.

3.3 Keyword Indexes

The *signature file* has proved to be a convenient indexing technique for text data and multiattribute retrieval [23, 24, 25, 26, 27, 28]. Multidimensional index structures such as K-D-B tree, grid file [29], R-tree, and R*-tree are not appropriate for indexing text data represented by keywords, because they assume that the dimensionality of the domain space, which can be interpreted as the number of keywords in the case of keyword indexing, is small and constant. The number of keywords given by users to query image databases can be variable. Moreover, most of the multidimensional index structures suffer from the dimensionality curse. Therefore, we chose the signature file technique as our indexing method for keywords.

The main idea of the signature file is to derive properties of data objects, called *signatures*, and store them in a separate file. A collection of the derived signatures is called signature file. Signatures are hash-coded binary words of fixed length. In general, all bits of the signature are initially cleared to *null*, and then a hash function is applied to the object's values to determine which bits are set to *one*. A lot of research has been done on the improvement of the performance of a signature file [25, 26, 27]. However, most of the researches have been performed for *static* environments where update operations are rarely occurred. Our index design scheme requires a *dynamic* data environment, which means that the signature file must be allowed to grow and shrink. The two representative dynamic signature files are S-tree [24] and Quick filter [28]. The main idea of the S-tree is to group adjacent signatures in pages and build a B-tree on top of them to provide direct access to the leaf signature pages. However, the major problem of the S-tree is that the performance is degenerated as the query signature weight becomes lower.

The number of 1's in a signature is called the *signature weight*. In the Quick filter, a signature file is partitioned by a hash function and the partitions are organized by linear hashing. Therefore, it is appropriate for the dynamic environment where updates are occurred frequently and results in good performance in the queries with high signature weights. However, if the distribution of signatures is nonuniform, then similar signatures are frequently generated and therefore the overflow rate increases and the storage utilization decreases. These degenerate the performance of the Quick filter.

To attack these performance disadvantages of existing dynamic signature files, we combine the concepts of the HG-tree and the frame-sliced signature file [26]. Using the HG-tree, which is a complete dynamic index structure, we solve the problem caused by high overflow and low storage utilization. We also tackle the problem caused by light weight signatures through adopting of frame-sliced signature method. The leaf nodes of the HG-tree are built using the concept of the frame-sliced signature file. The directory regions in the nodes are represented by the image signatures.

At first, a signature is divided into *s frames*, and *c* frames are selected out of a total of *s* frames using one hash function h_1. To make up the *word signature* (i.e., the signature corresponding one keyword) *m* bits are set to "1" in the selected *c* frames using the second hash function h_2. The *frame signature* is constructed by superimposing the parts belong to the corresponding frame of word signatures. At last, the *image signature* describing the content of an image is constructed by combining the frame signatures.

Example 2. Figure 5 shows the procedure of constructing an image signature based on the frames when an image consists of a set of four keywords, {"sky", "sea", "bridge", "ship"}. In Figure 5, it is assumed that the length of an image signature is 12 bits, the number of total frames is 3, the number of frames to be selected is 2, and the number of bits to be set is 2. The word signature of keyword "sky", "sea", "bridge" and "ship" are assumed to be 0010 1000 0000, 0000 0100 0100, 0100 0001 0000 and 0010 0000 1000, respectively. The image signature becomes 0110 1101 1100 when we concatenate the frame signatures 0110, 1101 and 1100.

The leaf and internal node structures of the HG-tree indexing keyword signatures are slightly modified to accommodate the keyword indexes. The entry in a leaf node has the form (*oid*, *F*), where *oid* is a pointer to the raw image in the database and *F* is an image signature which consists of *s* frames (F_1, F_2, ..., F_s). The entry in an internal node has the form (*ptr*, *S*), where *ptr* is a pointer to the child node and *S* is a signature made by superimposing all the image signatures in the corresponding child node.

a set of keywords A_k = { “sky”, “sea”, “bridge”, “ship” }

keywords	frame1	frame2	frame3
sky	0010	1000	
sea		0100	0100
bridge	0100	0001	
ship	0010		1000
image signature	0110	1101	1100

Figure 5. The procedure constructing the image signature

When we construct the HG-tree with image signatures, the values of the image signatures are interpreted as the Hilbert values and they are inserted in the order of Hilbert values. This interpretation procedure corresponds to the mapping a point in *s*-dimensional space into a point in a linear Hilbert curve. The number of frames, *s*, constituting an image signature determines the dimensionality of the keyword domain space and the image signature used in a node of the HG-tree determines a directory region in the domain space. The reason of using Hilbert mapping instead of simply concatenating *s* frames is to place the similar signatures into the same disk pages. In general, when the similar signatures are placed in the same page, they have a high qualifying probability once the page has been designated by the search procedure.

Example 3. Figure 6 show the binary number sequence and the Hilbert number sequence in a 4×4 grid space. Let us consider the ordering with binary numbers in Figure 6(a). Since the two points B_1 and B_2 are adjacent with respect to the distance, the corresponding signatures of B_1 and B_2 may be quite similar when we concatenate the codes in two directions. The signatures (or numeric values) of B_1 and B_2 are 0011 and 0111, respectively, and the signature suffixes are very similar. However, the probability of placing the two signatures into same page is very low, because the numeric difference of two signatures is relatively large. This may result in many random accesses on disk. On the other hand, The signatures (or Hilbert values) of H_1 and H_2, which are adjacent points in the domain space, are 0101 and 0110 , respectively. Therefore, the probability of placing the two signatures into same page may be very high, because the numeric difference of two signatures is only 1. The points close on the Hilbert curve are also close in the domain space as you see in Figure 6(b). This characteristic can achieve a better clustering of pages and can avoid expensive random disk accesses. The characteristics of good clustering and high storage utilization (above 80% in average) of the HG-tree achieve good query performance on the keyword indexes as well.

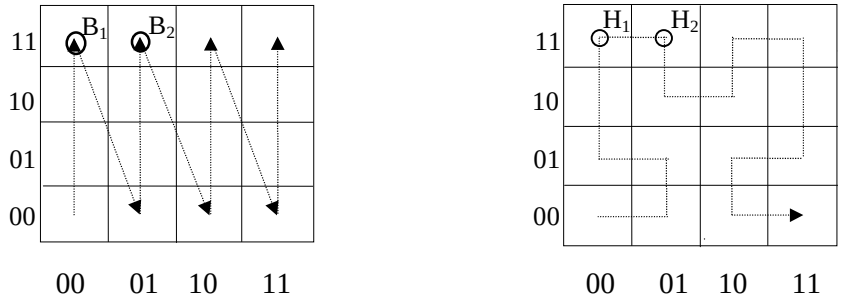


Figure 6. Illustration of binary number sequence vs. Hilbert number sequence

4. Processing of Complex Similarity Queries

The result of a *k*-nearest neighbor query based on the visual feature values is a list with *k* images sorted by similarity. On the other hand, the results of queries based on the semantic features and keywords are almost sets of images which satisfy the queries. When the complex similarity queries are issued to retrieve more exact matches, the search procedure must synthesize the results of different types of queries in a certain consistent manner.

Example 4. Let us consider the query Q_1 where a user wants to retrieve 5 most similar images to a given image *I* with respect to color histogram and whose *subject* is '*animal'*. Then, the query predicate can be represented as follows:

$$(A_v(\textit{color histogram}) = \textit{'I'}) \wedge (A_s(\textit{subject}) = \textit{'animal'})$$

In this case, the query result is probably a list with 5 objects sorted by the visual similarity to the image *I*, where the value of *subject* attribute in A_s is *animal*.

A reasonable way to evaluate these types of queries, in which the result of one query is a set and the result of the other query is a sorted list, would be to first evaluate the query whose result becomes a set, and then to find the sorted list consisting of the required number of objects from the set. Therefore, to evaluate the query Q_1 in Example 1, we determine all objects that satisfy the predicate A_s(s*ubject = 'animal'*), and then obtain similarity scores of the objects, and finally return 5 objects which have highest similarity scores.

Example 5. Let us now consider the query Q_2 with two predicates where a user wants to retrieve *k* best matches which are similar to both images I_1 and I_2 with respect to color histogram:

$$(A_{v1}(\textit{color histogram}) = \textit{'}I_1\textit{'}) \wedge (A_{v2}(\textit{color histogram}) = \textit{'}I_2\textit{'})$$

There may be several ways to deal with this kind of complex similarity query. An obvious fact

is that it is not correct to retrieve the best match only for single predicate, because the best match for the overall query needs not to be the best match for the single predicate. In this kind of query where several sample images are given, we can transform it to the query with only one predicate containing an image which is best matched to all given query images. In the query Q_2, we can find a point P which is closest to both points P_1 and P_2 corresponding to the images I_1 and I_2, respectively. Then we retrieve k objects closest to the point P.

Another solution to this kind of query is Fagin's A_0 algorithm [8], which independently evaluates the predicates in the query and computes the overall similarity scores of the objects in each result set based on a certain rule combining the similarity scores. Finally it returns k objects in the order of highest similarity scores.

It should be noted that the query results from the above two methods, i.e., the method transforming the features in several images into the features of one image in advance and Fagin's A_0 algorithm, may be different. In the former method, the search target is changed to the object most common to the given query objects, on the other hand, in the latter method, the most similar objects among the separately selected best matches are chosen. The difference of search methods may make the final results different. To determine which method produces more exact results is not trivial and can be dependent on the user's viewpoint. One obvious fact is that the former is far more efficient than the latter. Because there is only one search to the index in the former and on the other hand, there are many index searches as the number of conjuncts in the latter.

Example 6. Consider an image database where images are characterized globally by their dominant colors and textures, and also characterized by shapes and locations of the objects in the image. Let us assume that two set of visual feature values, i.e., A_{v1} = {color, texture}, A_{v2} = {shape, location}, are indexed separately. Let us consider the following query Q_3:

$$(A_{v1}(color, texture) = `I_1') \wedge (A_{v2}(shape, location) = `I_1').$$

In this case, since the domain space of two predicates are made different, the evaluation of each predicate must be performed independently and the algorithm such as Fagin's A_0 should be incorporated to combine two result sets. However, it is important to note that the visual features should be integrated in a single multidimensional index if possible, since the independent evaluation of the predicates and the combination of the separate partial results are very expensive.

Fagin used the standard rules of fuzzy logic [30] for evaluating Boolean combinations of atomic formulas:

Conjunction rule: $s_{A \wedge B}(x) = \min \{s_A(x), s_B(x)\}$

Disjunction rule: $s_{A \vee B}(x) = \max \{s_A(x), s_B(x)\}$

Negation rule: $s_{\neg A}(x) = 1 - s_A(x)$

where $s_A(x)$ is the similarity score of object x under the query A. Although these rules have some attractive points as shown in [8], the obvious drawback is that it depends on only the single operand, that is, $s_{A \wedge B}(x)$ (as well as $s_{A \vee B}(x)$) is always equal either to $s_A(x)$ or $s_B(x)$. This does not reflect the effect of all predicates. With this observation, we use other combination rule to synthesize the partial results evaluated independently, which will depend on all predicates in the query.

The first step towards an effective and efficient evaluation of complex similarity queries concerns how to compute the similarity score. The next step is how to combine the similarity scores when they are computed independently. As mentioned before, it is desirable to build a single multidimensional index which indexes all visual features if possible rather than to build multiple indexes. However, the method to synthesize other types of queries should be developed in the cases that data reside in multiple systems or features describing the image have inherently different characteristics, e.g., some features are quantifiable and the others are not.

In general, evaluating the similarity of an object with respect to a query value can be done through the *distance function* measuring the distance between feature values. The high similarity score corresponds to low distance and low similarity score to high distance. Definition 1 gives the distance function used in our work.

Definition 1. Distance function

A *distance function*, d, for any pair of feature values (x, y) from the domain space D yields a non-negative real value between 0 and 1, which shows the normalized distance between x and y.

$$d: D^2 \rightarrow [0, 1]$$

The 0 distance between a pair of feature values denotes the exact match and the 1 distance shows the maximum difference.

Although, in principle, arbitrary distance functions can be used, we assume that the distance function d is a *metric*, that is, for each triple of values $x, y, z \in D$ the followings are hold:

1. $d(x, y) = d(y, x)$. (symmetry)
2. $d(x, y) \geq 0$ and $d(x, y) = 0$ iff $x = y$. (positivity)
3. $d(x, y) + d(y, z) \geq d(x, z)$. (triangle inequality)

The reason for adopting the metric distance function is that we use the index structures which index the data over the metric space. Definition 2 gives the similarity function which assigns maximum similarity (i.e., 1) in case of 0 distance and makes similarity inversely related to distance.

Definition 2. Similarity function

We define the *similarity function*, *s*, for any pair of feature values (*x*, *y*) as

$$s(x, y) = 1 - d(x, y),$$

where *d* is a distance function. The similarity score is also a real value in the range [0, 1].

Let us now consider how to combine the similarity scores computed in independent predicate evaluations. Fagin used the standard rule of fuzzy logic for this purpose. Other models such as "algebraic" semantics for logical operators and "list-of-keywords" model may be used [7]. Although it is not clear which is the best combination rule, it is important to realize that any specific model has some advantages and drawbacks.

We do not consider the Fagin's rule as our combination method, because, in that rule, the best match only for one predicate determines the overall best match. If we normalize the size of domain space to 1, we can interpret the logical statement A $\wedge$ B as the probability of satisfying two conjuncts A and B simultaneously. That is, the similarity score of A can be interpreted as the probability of how A is similar to a given query image. Similarly, the similarity score of A $\wedge$ B can be interpreted as the probability of how A and B are similar to a given query image simultaneously. Thus we use the *probability function* as a rule to combine independent partial similarity scores. Then the overall similarity score is determined by the following definition and is always in the range of [0, 1].

Definition 3. Overall similarity function *s*

$$\text{overall similarity score of object } x = s_{A1\wedge\ldots\wedge Am}(x) = \prod_{i=1}^{m} s_i(x)$$

where s_i the similarity function over the *i*-th predicate *Ai* and *m* is the number of conjuncts, i.e., the number of similarity predicates.

It can be easily shown that this function is *monotonic increasing*, that is, $s_{A1\wedge\ldots\wedge Am}(x) \le s_{A1\wedge\ldots\wedge Am}(x')$ when $s_{Ai}(x) \le s_{Ai}(x')$ for every *i*. Intuitively, if the similarity score of object *x′* under the query *Ai*, $1 \le i \le m$, is at least as big as that of object *x*, the similarity score of object *x′* under the query $A1\wedge\ldots\wedge Am$ is at least as big as that of object *x* under the query $A1\wedge\ldots\wedge Am$.

With these distance and similarity functions, the algorithm **ComplexNNSearch** based on

the Fagin's A_0 algorithm for processing the complex *k*-nearest neighbor query where the similarity scores over each predicate should be found independently is given below. Note that the procedure **NNSearch(A_i, *k*)** receives the predicate A_i and the number, *k*, of objects to retrieve, and returns *k* objects with highest similarities. It can also output the next *k* similar objects from the previously stopped position with the set of remaining objects.

Algorithm ComplexNNSearch(*m*, *A*, *k*)

// *m* is the number of conjuncts (or predicates) in the query,
// *A* is the set consisting of *m* predicates A_i, i.e., $A = \{A_i\}$, $i = 1..m$
// *k* is the number of objects to retrieve.

1. [Initialize]
   for $i \leftarrow 1$ to *m* do {
      $X^i \leftarrow$ NULL // X^i is initialized to empty set. X^i will contain the objects
                  // returned from the *k*-nearest neighbor search.
   }
2. [Find the *k* matches]
   do {
      for $i \leftarrow 1$ to *m* do {
         $G^i \leftarrow$ **NNSearch(A_i, *k*)** // G^i is the set which will contain the objects and their
                  // similarity values in the output from the processing of
                  // the *i*-th *k*-nearest neighbor search conjunct A_i.
         $X^i \leftarrow X^i \cup G^i$
      }
      $L \leftarrow \bigcap_{i=1}^{m} X^i$ // *L* is a set that has the intersection of X^is.
   } while ($|L| < k$) // Perform loop until *L* has at least *k* objects.
                  // $|L|$ denotes the number of objects in the set *L*.
3. [Compute the similarity scores for the candidates]
   $Y \leftarrow \{ x \mid x \in \bigcup_{i=1}^{m} X^i \}$
   $j \leftarrow$ the number of objects in *Y*
   for $i = 1$ to *j* do {
      Compute the similarity score c_i of x_i in *Y* using the overall similarity function.
      Restore (x_i, c_i) into *Y*.
   }
4. [Find the most similar *k* objects]
   Return the sorted list of *k* objects with highest similarity scores.

5. Indexing and Retrieval Mechanism for Complex Similarity Queries

To sum up the discussion to this point, the indexing and retrieval mechanism that integrates the three types of indexes is shown in Figure 7.

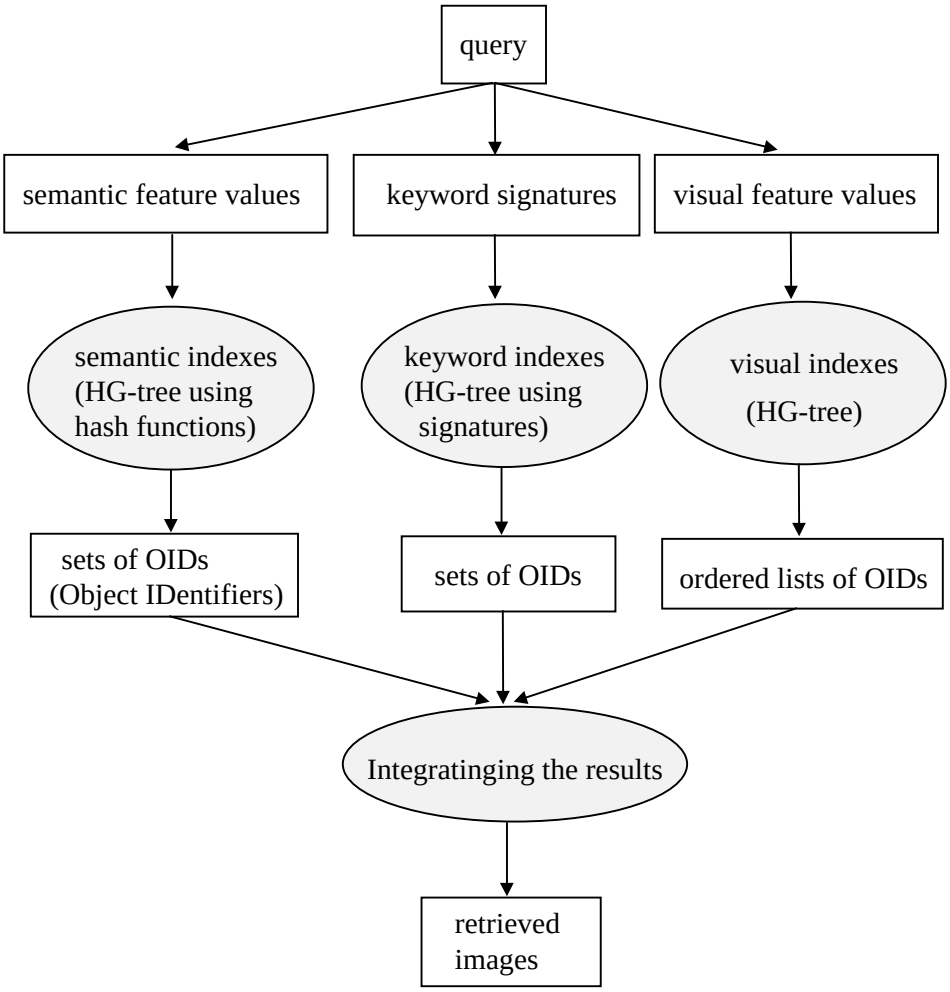


Figure 7 Indexing and Retrieval Mechanism for Complex Similarity Queries

The predicates of the query are interpreted to appropriate feature values, i.e., visual, semantic, and keywords, during query processing. These feature values are evaluated and searched through the corresponding indexes. Finally, Object IDentifiers (or OIDs), are acquired and resolved to answer the query. Although we model the image with three types of features and index the image database with the corresponding features, other kinds of features and indexes can be easily incorporated into our mechanism without any change of our basic concepts and framework.

6. A Prototype Image Retrieval System

A prototype content-based image retrieval system has been implemented. The architecture of the system is shown in Figure 8. The system consists of the components described in the above section. The system supports complex similarity queries which consist of semantic feature-based, visual feature-based, keyword-based predicates.

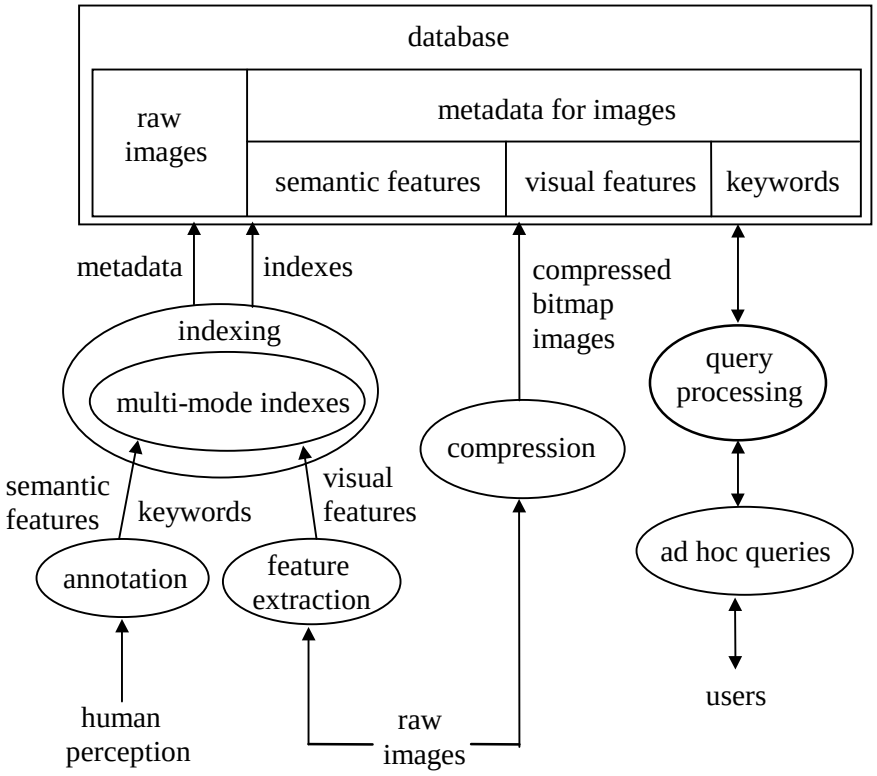


Figure 8 Architecture of content-based image retrieval system

6.1 The Image Database

To test the effectiveness of our indexing and query processing mechanism for content-based image retrieval, we have constructed an image database that has a 1,064 images. The images are 256-color bitmaps with a variety of characteristics. In the database, images are stored together with three kinds of attributes, i.e., semantic features, visual features, and keywords, so that images can be retrieved from these descriptions. The visual features are extracted automatically by image processing subsystem and the semantic features and keywords are inserted manually in database population time.

6.2 Visual Feature Extraction

To acquire the visual features that characterize images we used statistical color moments of the histogram of the image because color has excellent discrimination power in image retrieval system. Since most histogram bins of an image are sparsely populated and only a small number of bins have the majority of pixel counts, we used only the largest 32 bins (in terms of pixel counts) as the representative bins of the histogram. We used first two moments of the histogram as descriptors of an image:

$$\mu_i = \frac{1}{n}\sum_{j=1}^{k} f_{ij}x_{ij}, \qquad i = 1, 2, 3$$

$$\sigma_i = \left(\frac{1}{n}\sum_{j=1}^{k} f_{ij}\left(x_{ij} - \mu_i\right)^2\right)^{\frac{1}{2}}, \quad i = 1, 2, 3$$

where x_{ij} is the value of color component of the j-th bin, f_{ij} is the frequency of x_{ij}, k is the number of total bins, i.e. 32, and n is the total number of pixels in the histogram. Since we used the RGB color model, the i-th color component corresponds to one of red, green, and blue. The first moment, μ_i, defines the average intensity of each color component. The second moment, σ_i, is a measure of contrast that can be used to establish descriptors of relative smoothness.

Measures of global color statistics using only histograms suffer from the limitation that they carry no information regarding the relative position of pixels. To overcome this limitation to some extent, we divided the image into 4 sub-areas and computed 2 moments for each sub-area, resulting in a 24 (= 2 moments × 3 color components × 4 sub-areas) visual features for an image.

Using this 24-dimensional feature vector, we estimate the similarity, $s(S, T)$, between two color histograms S and T as follows:

$$s(S, T) = 1 - \frac{\sum_{k=1}^{4}\left(\sum_{i=1}^{3}\left(\left|\mu_{ik}(S) - \mu_{ik}(T)\right| + \left|\sigma_{ik}(S) - \sigma_{ik}(T)\right|\right)\right)}{\Delta},$$

where Δ is a normalizing factor.

6.3 Performance Experiments of the HG-tree

To assess the performance of the HG-tree we created six groups of 24-dimensional data files that contained different distributions of data as in [31]: uniform, diagonal, bit, parallel, clustered distributions and real data. Five synthesized data files contain 10,000 objects without duplicates. The real data files contain 1,064 256-color images with a variety of image contents.

To demonstrate the performance for queries we generated six groups of range queries and

eight groups of nearest neighbor queries: The regions of the six groups of range queries are squares varying in size which are 0.001%, 0.01%, 0.1%, 1%, 10%, and 20% of the whole data space and their centers are uniformly distributed in the data space. The *k*'s of eight groups of *k*-nearest neighbor queries are 1, 10, 20, 40, 60, 80, 100, and 120. For each experiment, 1,000 randomly generated queries were asked and the results were averaged. We chose the page size for leaf nodes and internal nodes to be 1 Kbyte.

We compared the HG-tree against the buddy-tree, because Seeger and Kriegel [31] reported that the buddy-tree is the best one among multidimensional point index structures with respect to the average range query performance. For all operations, we have measured the number of disk accesses per operation.

We report the average number of disk accesses per query in Tables 1 and 2 for range and nearest neighbor queries. For the sake of an easier comparability, we have normalized the average number of disk accesses for range and nearest neighbor queries in the buddy-tree to 100% in each table. Summarizing the outcome of our comparison, we can state that the HG-tree exhibits on the average 20-30% better range and nearest neighbor query performances than the buddy-tree. More detailed performance comparison appears in [1].

Table 1. Average over all types of range queries depending on distributions

	uniform	clustered	bit	parallel	diagonal	real
buddy-tree	100.0	100.0	100.0	100.0	100.0	100.0
HG-tree	77.2	71.1	69.2	73.2	61.2	57.8

Table 2. Average over all types of nearest neighbor queries depending on distributions

	uniform	clustered	bit	parallel	diagonal	real
buddy-tree	100.0	100.0	100.0	100.0	100.0	100.0
HG-tree	88.8	77.8	80.4	88.9	72.1	61.4

6.4 Sample *k*-Nearest Neighbor Queries

Figures 9 and 10 show the results of two 12-nearest neighbor queries Q1 and Q2:

(a) Q1: Find top 12 images similar to the image *golden2.jpg* with respect to color and texture.

(b) Q2: Find top 12 images similar to the image *tigera4.bmp* with respect to color and texture.

The images on the upper-left corner in Figure 9 and 10 are the query images and 12 most similar images are retrieved in left-right and top-down sequence: In Figure 9, *golden2.jpg* is the

query image and, of course, it is the most similar image, *golden5.jpg* is the second similar image, and *beach2.jpg* is the 12*nd* similar image. The results of Q1 and Q2 are obtained only using visual features (i.e., color and texture). Obviously, all images retrieved from a real image database have similar color properties to the given query image.

Figure 11 is the result of 12-nearest neighbor query with query image *tigera4.bmp* and subject = '*animal*'. Compared with the result in Figure 10, the result in Figure 11 shows that the images similar to *tigera4.bmp* are retrieved from among the images whose subject is *animal*. Figure 12 shows the top 12 images which have subject = '*flower*' and keyword = {'*yellow*'}, and are similar to the query image *fyello3.jpg*.

As shown in the above examples, the more features we specify in the query, the more discriminating power increases. However, as the number of features to be indexed increase, the dimensionality of the domain space increases, and then eventually, the search cost increases.

7. Conclusions

In this paper, we have considered the issues concerned with the content model, the indexing scheme, and the processing for complex similarity queries as a whole in content-based image retrieval systems. The content of an image consists of a set of visual features, a set of semantic features, and a set of keywords. Each component of the content may have different structural characteristics. To index three types of feature sets, three types of index structures are proposed. The underlying index structure is the HG-tree. The performance advantages of the HG-tree makes our indexing scheme efficient. The techniques for processing complex similarity queries are also provided. These three issues are very important things to construct the effective and efficient image retrieval system. Finally, we have demonstrated the effectiveness of our indexing and retrieval mechanism by implementing prototype image retrieval system which can support a wide range of queries.

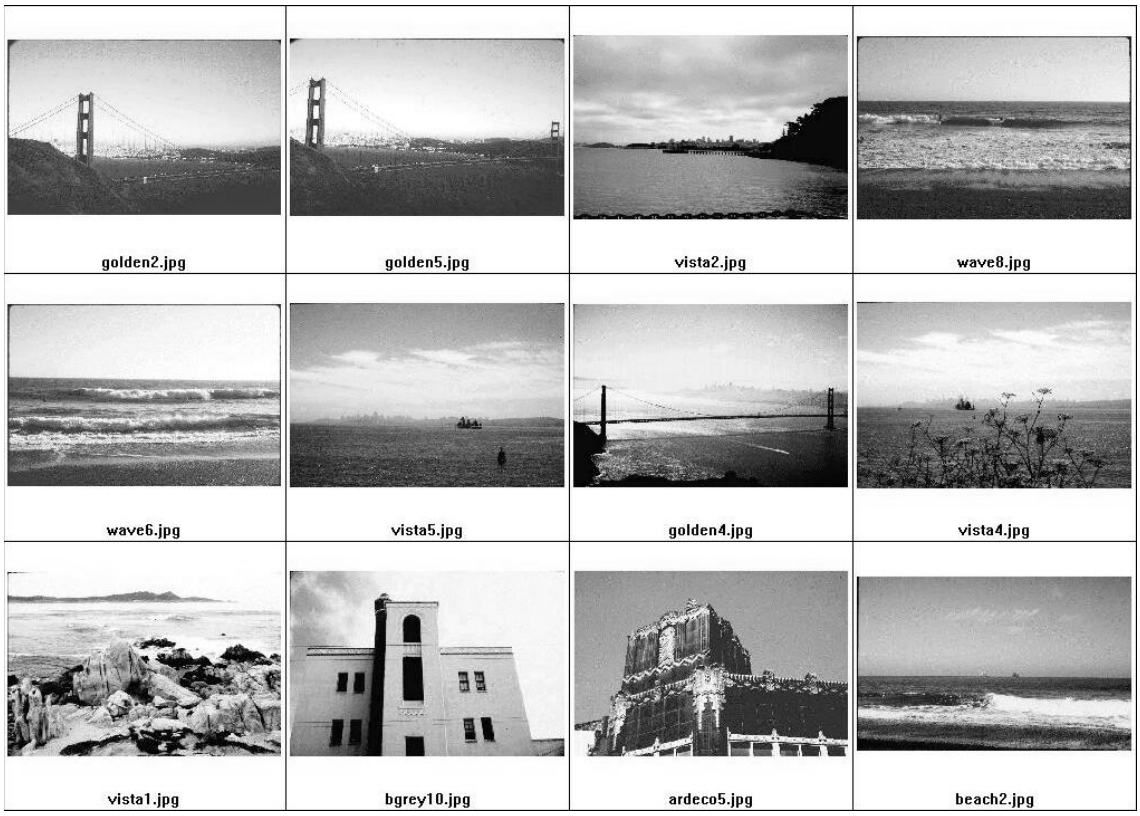


Figure 9. Result of 12-nearest neighbor query: Top 12 images similar to *golden2.jpg* with respect to color and texture are retrieved.

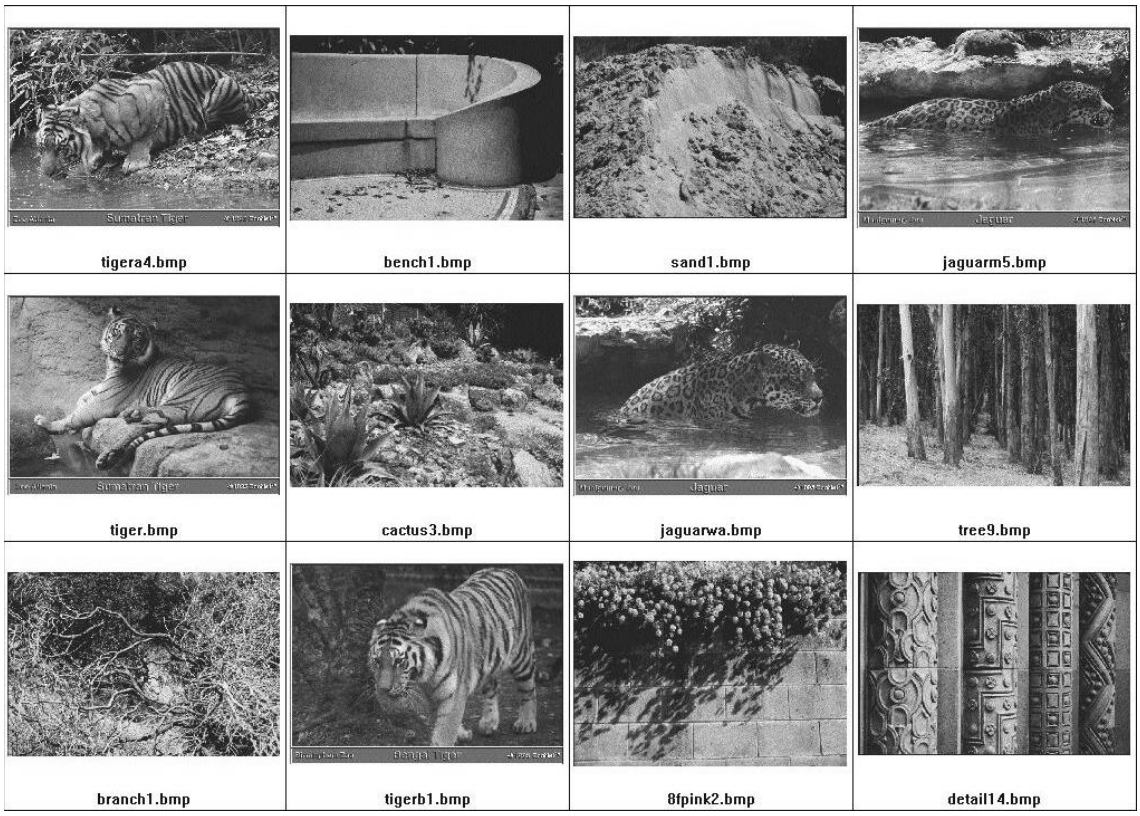


Figure 10. Result of 12-nearest neighbor query: Top 12 images similar to *tigera4.bmp* with respect to color and texture are retrieved.

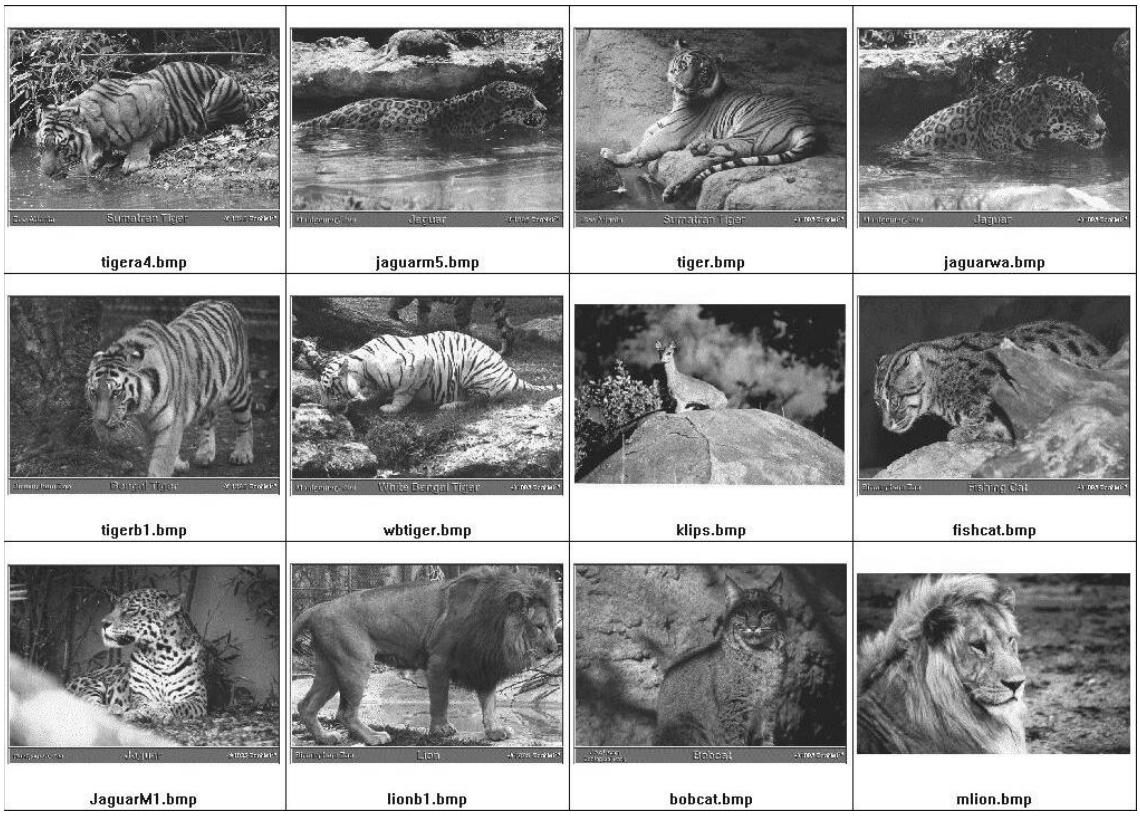


Figure 11. Result of 12-nearest neighbor query: Top 12 images similar to *tigera4.bmp* with respect to color and texture and whose subject = '*animal*'.

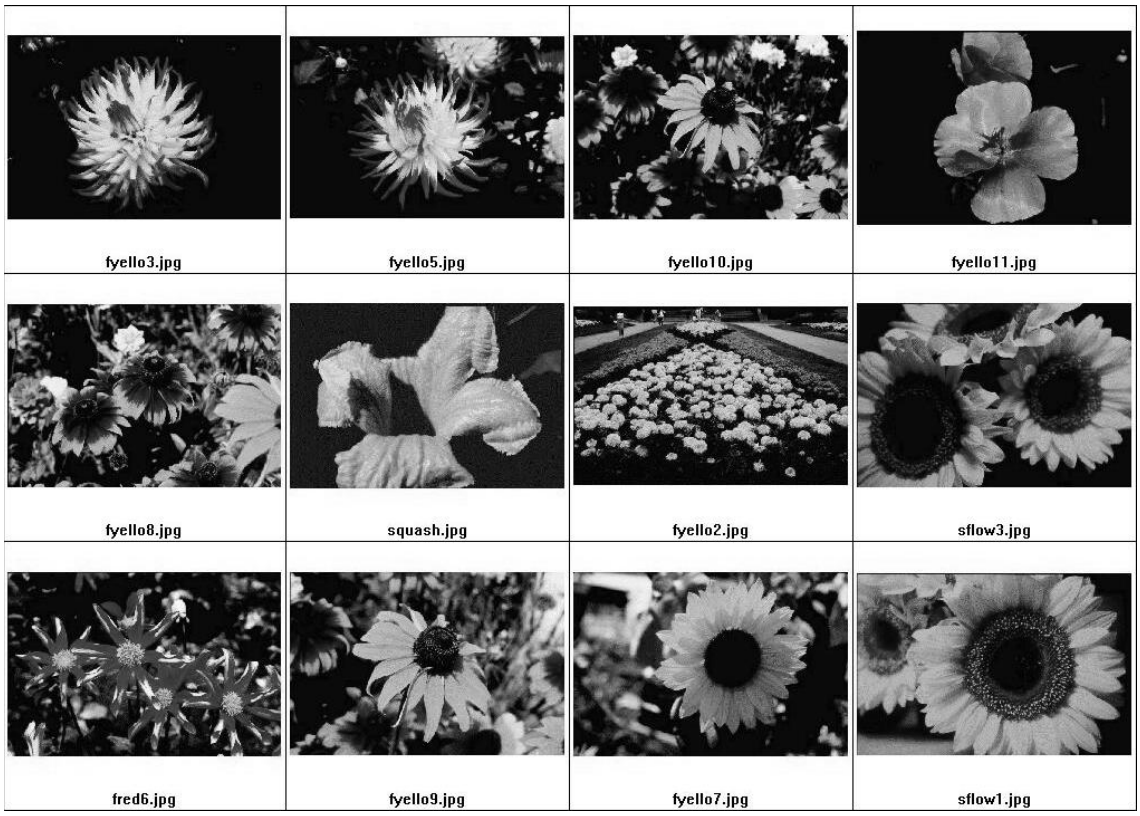


Figure 12. Result of 12-nearest neighbor query: Top 12 images similar to *fyello3.jpg* with respect to color and texture and whose subject = '*flower*' and keyword = {'*yellow*'}.

References

1. G.-H. Cha and C.-W. Chung, "A New Indexing Scheme for Content-Based Image Retrieval," *Multimedia Tools and Applications*, 6(3), May 1998, 263-288.
2. M. Flickner et al., "Query by Image and Video Content: The QBIC System," *IEEE Computer*, September 1995, 23-32.
3. V.E. Ogle and M. Stonebraker, "Chabot: Retrieval from a Relational Database of Images," *IEEE Computer*, September 1995, 40-48.
4. J.K. Wu, Y.H. Ang, P. Lam, H.H. Loh, and A.D. Narasimhalu, "Inference and retrieval of facial images," *Multimedia Systems*, 2(1), 1994, 1-14.
5. J.K. Wu, A.D. Narasimhalu, B.M. Mehtre, C.P. Lam, and Y.J. Gao, "CORE: a content-based retrieval engine for multimedia information systems," *Multimedia Systems*, 3(1), 1995, 25-41.
6. D. Comer, "The Ubiquitous B-tree," *ACM Computing Surveys*, 11(2), 1979, 121-137.
7. P. Ciaccia, M. Patella, and P. Zezula, "Processing Complex Similarity Queries with Distance-based Access Methods, HERMES Technical Report 22, September 1997.
8. R. Fagin, "Combining Fuzzy Informatin from Multiple Systems," *Proceedings of the 15th ACM Symposium on Principles of Database Systems*, 1996, 216-226.
9. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles," *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1990, 322-331.
10. E.G.M. Petrakis and C. Faloutsos, "Similar Searching in Large Image Databases," Technical Report CS-TR-3388, University of Maryland, 1994.
11. A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1984, 47-57.
12. R. Mehrotra and J.E. Gary, "Similar-Shape Retrieval in Shape Data Management," *IEEE Computer*, September 1995, 57-62.
13. J. T. Robinson, "The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1981, 10-18.
14. H.J. Zhang and D. Zhong, "A Scheme for Visual Feature based Image Indexing," *Proceedings of the IS&T/SPIE Conference on Storage and Retrieval for Image and Video Databases III*, 1995, 36-46.
15. T. Kohonen, "The Self-Organizing Map," *Proceedings of the IEEE*, 78(9), September 1990, 1464-1480.

16. T.-C. Chiueh, "Content-Based Image Indexing," *Proceedings of the 20th VLDB Conference*, 1994, 582-593.
17. S. Brin, "Near Neighbor Search in Large Metric Spaces," *Proceedings of the 21st VLDB International Conference*, 1995, 574-584.
18. D. Hilbert, "Uber die stetige Abbildung einer Linie auf ein Flachenstuck," *Math. Annalen*, 38, 1891.
19. G. Peano, "Sur une courbe qui remplit toute une aire plane," *Math. Annalen*, 36, 1890, 157-160.
20. C. Faloutsos, "Gray codes for partial match and range queries," *IEEE Transactions on Software Engineering*, 14(10), 1988, 1381-1393.
21. C. Faloutsos and S. Roseman, "Fractals for secondary key retrieval," *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1989, 247-252.
22. Jagadish, "Linear Clustering of Objects with Multiple Attributes," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1990, 332-342.
23. S. Christodoulakis and C. Faloutsos, "Signature files: An access method for documents and its analytical performance evaluation," *ACM Transactions on Office Information Systems*, 2(4), October 1984, 267-288.
24. U. Deppisch, "S-Tree: A Dynamic Balanced Signature Index for Office Retrieval," *Proceedings of the ACM Conferences on Research and Development in Information Retrieval*, September 1986, 77-87.
25. D.L. Lee and C.-W. Leng, "Partitioned Signature Files: Design Issues and Performance Evaluation," *ACM Transactions on Office Information Systems*, 7(2), 1989, 158-180.
26. Z. Lin and C. Faloutsos, "Frame-Sliced Signature Files," *IEEE Transactions on Knowledge and Data Engineering*, 4(3), June 1992, 281-289.
27. R. Sacks-Davis, A. Kent, and K. Ramamohanarao, "Multikey Access Methods Based on Superimposed Coding Techniques," *ACM Transactions on Database Systems*, 12(4), December 1987, 655-696.
28. P. Zezula, F. Rabitti, and P. Tiberio, "Dynamic Partitioning of Signature Files," *ACM Transactions of Information Systems*, 9(4), October 1991, 336-369.
29. J. Nievergelt, H. Hinterberger, and K.C. Sevcik, "The grid file: an adaptable, symmetric multikey file structure," *ACM Transactions on Database Systems*, 9(1), 1984, 38-71.
30. L.A. Zadeh, "Fuzzy Sets," *Information and Control*. 8, 1965, 115-124.
31. B. Seeger and H.-P. Kriegel, "The Buddy-Tree: An Efficient and Robust Access Method

for Spatial Data Base Systems," *Proceedings of the 16th International Conference on VLDB*, pp. 590-601, 1990.