2-D-S tree: An index structure for content-based retrieval of images

Youping Niu M. Tamer Özsu Xiaobo Li

Department of Computing Science, University of Alberta, Edmonton, Canada T6G 2H1

ABSTRACT

An important feature to be considered in the design of multimedia DBMSs is content-based retrieval of images. Most work in this area has focused on feature-based retrieval; we focus on retrieval based on spatial relationship, which include directional and topological relationships. The most common data structure that is used for representing directional relations is the 2-D string. The search process, however, is sequential and the technique does not scale up for large databases. We propose a new indexing structure, the 2-D-S-tree, to organize 2-D strings for query efficiency. The 2-D-S-tree is completely dynamic; inserts and deletes can be intermixed with searches and no periodic reorganization is required. A performance analysis is conducted, and both analytical analysis and experimental results indicate that the 2-D-S-tree is an efficient index structure for content-based retrieval of images.

Keywords: Image indexing, multimedia DBMSs, 2D-string

1. INTRODUCTION

The penetration of database management technology into the domain of multimedia information systems poses interesting challenges. Providing database functionality over multimedia data (i.e., text, images, audio, and video) requires significant extensions to the traditional database management system (DBMS) technology. In this paper, we focus on querying over image data. This class of systems are generally known as image DBMSs.

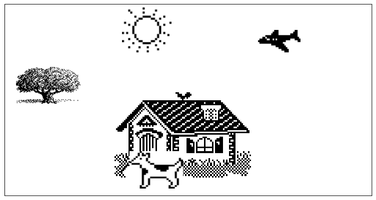
Traditionally, image DBMSs facilitate querying based on features. The most common features are *texture*, *color*, and *shape*.[1–5] The user identifies a sample image and the system searches the database for all images that are "similar" to the sample image according to user specified levels of texture, color and shape metrics. This type of querying is well understood[6] and there are number of systems that support it, e.g., Virage,[7] QBIC.[1]

Another type of image querying is based on spatial relationships of the *salient* objects in the images. Spatial relationships, such as relative positioning, adjacency, overlap, and containment, enable users to ask queries of the type "show all the images where a *car* is to the left of a *building*." Systems that couple spatial and feature-based querying enable sophisticated queries to be posed such as "show all the images where a *red car* is in front of a *building* that looks like a *sphere*." Spatial querying is not as well studied as feature-based querying.

In this paper, we propose a new content indexing structure for images based on the 2-D string representation[8] of the spatial relationships of their contents. 2-D string is the most commonly used data structure for spatial reasoning and spatial similarity computing. The spatial relationships in images, as well as the queries, are represented as 2-D strings. Thus query execution is reduced to 2-D string subsequence matching. The technique works well for queries that are posed on a single image (i.e., find subimages in a given image), but searches over a large set of images are quite inefficient since string matching is performed sequentially, making it inappropriate for large databases. One solution[9] to this problem is to develop a hash table for a set of ordered triples (a variant of 2-D strings) to facilitate the search. However, this method can not distinguish query types and is restricted in the types of queries it can address (we define query types in Sect. 2). Furthermore, the hashing table needs periodic reorganization to maintain its performance once some insertions and deletions are performed. Other 2-D string variants[10–13] address the issues of efficient segmentation, storage space and reasoning complexity, but fail to overcome the fundamental drawback of sequential searching.

We propose an indexing structure, which we call 2-D-S-tree, that organizes 2-D strings efficiently for three types of ($type-0$, $type-1$ and $type-2$) querying. The 2-D-S-tree is completely dynamic; inserts and deletes can be

Further author information: E-mail: {niu,ozsu,li}@cs.ualberta.ca



	Sun (D)	Plane (E)
Tree (A)		
	House (B) Dog (C)	

Figure 1. (a) An image; (b) the symbolic image of (a).

intermixed with searches and no periodic reorganization is required. A performance analysis is conducted, and both analytical and experimental results indicate that the 2-D-S-tree is an efficient index structure for content-based retrieval of images.

The remainder of the paper is organized as follows. In Sect. 2, we provide an overview of spatial structures and review the basics of the 2-D string approach. Our new indexing scheme for 2-D strings is presented in Sect. 3. The performance analysis of our indexing scheme is presented in Sect. 4. Section 5 presents our conclusions and future work.

2. BACKGROUND

A large number of spatial data structures have been developed such as R* trees[14] and the rest of the R tree family,[15–17] *linear quadtrees*,[18] *grid-files*[19] and K-D-B-trees.[20] They can be used to store spatial data such as points, lines, and rectangles. In image DBMSs, however, we are particularly interested in the hierarchical representations of multi-dimensional point data since this hierarchical representation can be used as an index to efficiently retrieve data. There are a number of systems[1,21] which have used R* trees to index features such as texture, color, and shape by representing all these features as multi-dimensional points. In these systems, a typical query might be, "Find all images with color similar to a given image." The effectiveness of retrievals depends on dimensions of the indexed features and the efficiency of similarity measures. We are interested in spatial queries of the type "Find all images containing a cat to the right of a tree" and "Find all images containing a car in front of a house which is to the left of a tree." One can use the above discussed structures to answer these queries by performing a point by point (or rectangle by rectangle if the salient objects are represented in enclosing rectangles) check. However, a more efficient mechanism of analysis of the spatial relationships among salient objects is to represent them by higher level data structures which can support the spatial reasoning, flexible information retrieval, and visualization.

2-D strings[8] and its variants[10–13] are more appropriate constructs for the type of spatial queries that we would like to support. An image is processed using image processing and pattern recognition techniques, resulting in the segmentation of the image objects, and then labeled to produce a symbolic image. In our current work within the DISIMA project,[22] we first use edge detection, combined with other detection techniques, to draw the active contours of the isolated objects. We then label (automatically or manually) each salient object (interesting object) with a symbol which can identify it and this is used for indexing and querying. Finally, a 2-D string is automatically generated for the image. The whole process can be automatic or semi-automatic depending on the types of images. For example, the five objects in the original image in Figure 1(a) can be represented by the symbolic image in Figure 1(b). The spatial relationship between two objects is denoted by one of the relations, $\{=, <, :\}$, where the symbol "=" denotes the spatial relation "at the same location as", the symbol "<" denotes the spatial relations "left of/right of" and "below/above", and the symbol ":" denotes the spatial relation "in the same set as." These relationships are specified in both x and y dimensions resulting in a two dimensional string. The 2-D string representation of the symbolic image in Figure 1(b) is: $(tree < house : dog = sun < plane, house : dog < tree < sun = plane)$.

A query such as "show images which have a tree to the left of a house" can be represented by the string $(tree < house,\ tree < house)$ and the problem of content-based retrieval of images becomes one of the 2-D string subsequence matching.

Three types of 2-D string subsequences are defined. A string u is a $type-i$ one-dimensional (1-D) subsequence of a string v, if (1) u is contained in v, i.e., if u is a subsequence of a permutation string of v, and (2) $a_1w_1b_1$ is a substring of u and $a_2w_2b_2$ is a substring of v, a_1 matches a_2 in v and b_1 matches b_2 in v, then

$type-0$: $r(b_2) - r(a_2) \geq r(b_1) - r(a_1)$ or $r(b_1) - r(a_1) = 0$

$type-1$: $r(b_2) - r(a_2) \geq r(b_1) - r(a_1) > 0$ or $r(b_2) - r(a_2) = r(b_1) - r(a_1) = 0$

$type-2$: $r(b_2) - r(a_2) = r(b_1) - r(a_1)$

where $r(x)$, the *rank* of symbol x, is defined as one plus the number of "<" preceding the symbol x. The rank indicates the directional position of the symbol (object) in one dimension. For example, in Figure 1, along the x-axis, the rank of the tree is 1 and the rank of the house is 2.

These one dimensional subsequence matching operations can be easily to extended to two dimensions. Let (u, v) and (u', v') be the 2-D string representations of the symbolic images f and f', respectively. (u', v') is a $type-i$ 2-D subsequence of (u, v) if (1) u' is a $type-i$ 1-D subsequence of u and (2) v' is a $type-i$ 1-D subsequence of v. We say f' is a $type-i$ subimage of f.

When a query is mainly interested in finding all the images containing specified symbols (objects) with some exclusive spatial conditions, $type-0$ subsequence matching should be conducted. Given a key, for example, $a = b$, $type-0$ queries treat images with relationships $a = b$, $a : b$ and $a < b$ as satisfactory. But it excludes from the answer set images where $b < a$. For a query "Find all images containing a tree and a house as long as the house is not to the left of the tree", the image in Figure 1 would be retrieved as the result of a $type-0$ query.

A $type-1$ query is interested in finding all the images in which the symbols in the pattern maintain alignments between each other, but the "distance" between any two symbols is insignificant and may be different than in the original pattern. For example, "Find all images containing a *plane* to the right of a tree." An image containing a plane to the right of tree, but with a house between them, would be retrieved by a $type-1$ query.

A $type-2$ subsequence matching is more precise than both $type-0$ and $type-1$ subsequence matching. That is to say, if a local substring u_1 of u matches a local substring v_1 of v, then the local substring u_i of u must match the local substring v_i of v for any $i \geq 1$. Thus, the "distance" between objects is important. For example, $tree < plane$ is not a $type-2$ subsequence of $tree < house : dog = sun < plane$. Therefore, an image containing a plane to the right of tree but with a house between them would not satisfy a $type-2$ query: "Find all images containing a *plane* immediately to the right of a tree."

3. THE 2-D-S-TREE INDEXING

The 2-D string subsequence matching algorithm[8] sequentially checks if a 2-D string (u', v') is a $type-i$ 2-D subsequence of a 2-D string (u, v) for $i = 0, 1,$ or 2. The time complexity of the algorithm is $O(M) + O(N^2 * lp^3)$, where M and N denote the number of symbols contained in (u, v) and (u', v'), respectively, and lp is the maximum length of the matching tables built in the algorithm. For a query posed on a single image such as "Find all subimages in a symbolic image f that match a given symbolic image", the 2-D string approach may be efficient. However, in an image database, we are more often interested in queries such as, "Find all images containing a house with a lake on the east and a tree in the north" or "Find all images containing a house with a lake on the east or a tree in the north." In this case, the 2-D string specified for the query must match against all the 2-D strings in the database sequentially by using the subsequence matching algorithm. When the database only has a few hundred images, sequential matching may be acceptable. However, for a database with thousands of images, the sequential subsequence matching, even for a single query, has unacceptable overhead.

In this section, we develop a new index structure on top of 2-D strings to facilitate spatial searches using 2-D string approach. The index structure, we call the 2-D-S-tree, supports $type-0$, $type-1$ and $type-2$ queries efficiently. The 2-D-S-tree is completely dynamic; inserts and deletes can be intermixed with searches and no periodic reorganization is required.

3.1. The Structure of Leaf Node

The 2-D-S-tree is a set grouping index whose non-leaf nodes have a similar layout to the B^+ tree,[23] while the leaf node layout is different. Each non-leaf node in the 2-D-S-tree contains $q + 1$ pointers and q entries. Each pointer points to a child node, which is the root of a sub-tree of the 2-D-S-tree. Figure 2 shows the layout of a 2-D-S-tree leaf node. The fields of "overflow page" and "next page" are pointers pointing to the overflow page and the next node (sibling leaf node next to the current one), respectively. The key value here is a form of $A\theta B$, where A and B are symbols in V (V is a set of the vocabulary or symbols) and $\theta \in \{<, =, :\}$. Nodes correspond to disk pages if the

no. of records	overflow page	next page	key1	record of key1	key2	record of key2	......

Figure 2. Layout of the leaf node records of a 2-D-S-tree

indexes are disk-resident, and the structures are designed so that a search for a query requires visiting only a very small number of nodes. The indexes are completely dynamic; inserts and deletes can be intermixed with searches and no periodic reorganization is required.

For each key we define x_set (y_set), which contains all the elements in the form of $(id, rank, r\_off)$, where id is the identifier of a 2-D string which has the key as its $type-1$ subsequence along the $x-axis$ ($y-axis$), $rank$ is a set of pairs of ranks for the two symbols in the key in the 2-D string, and r_off is the offset of the rank along the $x-axis$ ($y-axis$)..

The internal structure of a record is defined as follows where x_off and y_off are the offsets of the sets of x_set and y_set, respectively:

```
struct xy_set {
        int              id;
        int              **rank;
        int              r_off;
};
struct Record {
        int              x_off;
        int              y_off;
        struct xy_set    *x_set;
        struct xy_set    *y_set;
};
```

Suppose we have the following two 2-D strings: $\{1, (A < C < B,\ B < A < C)\}$ and $\{2, (A < B = A < C < B,\ C = A = A < B < B)\}$ where 1 and 2 are the ids for the 2-D strings. For simplicity, we do not show the r_off for each rank, and x_off and y_off for each internal record in this and the following examples. Let's look at the record with the key: $A < B$. First, $x\_set = \{1, 2\}$ since $A < B$ is the subsequence in x-axis in both 2-D strings. That is $x\_set = \{1, 2\}$. Similarly, $y\_set = \{2\}$. However, ids by themselves cannot provide sufficient information to conduct the search. For example, since 1 is in x_set, we know that the first 2-D string contains $A < B$ as a subsequence in x-axis, but we don't know what the type of the subsequence is (here it is a $type-1$ subsequence). Thus, it is necessary to keep the information of rank for the symbols (in the key) in the sets of x_set and y_set as well. This information is a set of rank pairs in the form of $[i_x, i_y]$, where i_x and i_y are the corresponding rank of the first symbol and the second symbol in the key, respectively. Therefore, in x_set, the $rank$ for 1 is $r_1 = \{[1, 3]\}$, and the $rank$ for 2 is $r_2 = \{[1, 2], [1, 4], [2, 4]\}$. Thus, $x\_set = \{1, \{[1, 3]\}, 2, \{[1, 2], [1, 4], [2, 4]\}\}$, and similarly $y\_set = \{2, \{[1, 2], [1, 3]\}\}$. Actually the $rank$ set can be further simplified.

PROPOSITION 1. *For a given record with the key* $= A\theta B, \theta \in \{<, =, :\}$, *if* $[i_1, i_2]$ *and* $[i_3, i_4]$ *in the set of* r_n, *and* ($i_1 = i_3$ *and* $i_2 \le i_4$), *or* ($i_2 = i_4$ *and* $i_1 \ge i_3$), *then* $[i_3, i_4]$ *can be removed from* r_n.

Therefore, the pair $[1, 4]$ should be removed from r_2 in x_set, and the pair $[1, 3]$ should be removed from r_2 in y_set, to save space. Finally, for the key $A < B$, $x\_set = \{1, \{[1, 3]\}, 2, \{[1, 2], [2, 4]\}\}$ and $y\_set = \{2, \{[1, 2]\}\}$. The reason we can remove, for example, $[1, 4]$ from $\{[1, 2], [1, 4], [2, 4]\}\}$, from the $rank$ set is because $[1, 2]$ contains all the information we need, and $[1, 4]$ is redundant (since we know that if the key is a $type-2$ subsequence of the string, it must also be a $type-0$ or $type-1$ subsequence of the string). Notice that the remaining rank pairs for $id = 2$ in the x_set are necessary, and neither can be removed from the x_set. For example, for a $type-1$ query $(B = A < B,\ A < B < B)$, the second rank pair $[2, 4]$ will be needed, and for a $type-2$ query $(A < B,\ A < B)$, the first rank pair $[1, 2]$ will be used.

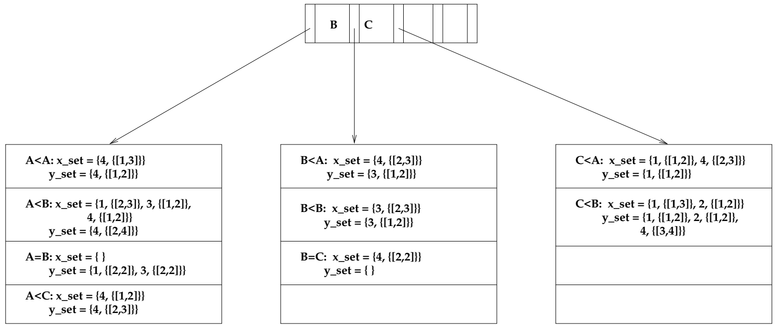


Figure 3. A 2-D-S-tree.

3.2. Insertions and Deletions

To insert (delete) a 2-D string into (from) a 2-D-S-tree, we first need to calculate all its possible subsequences (in the form of $A\theta B$), which will be used as the keys, and their associated rank pairs. Then, for each key, the procedures of insertion and deletion of a record are similar to those procedures of a B$^+$ tree. As a result, the *id* of the 2-D string, and its associated rank pairs, are added to (removed from) the leaf node of the 2-D-S-tree. A leaf node index record may be *small* (not larger than the size of an index page) or *large* (larger than the index-page size). A small index record can grow to a large index record, or simply grow out of bounds of its current index page. When a small index record grows out of the bounds of its index page, but remains a small record, the index page is split, and splits propagate up the tree. If an index record becomes a large record, an entire leaf node is assigned to it, and the part of the record that still does not fit in the node is stored in the overflow page(s).

The calculation of all possible keys and their associated rank pairs for a given 2-D string is rather complex. The algorithm first checks the relation symbol immediately next to the current symbol since different relationships are propagated and carried out to the following symbols in different ways. For example, '<' generates the keys with the current symbol as the first symbol and all the following symbols as the second symbol with '<' as the relationship between them, while ':' and '=' generate keys in different ways. Once a key and its associated rank pairs are generated, the procedure checks if the key is already generated. If so, the associated rank pairs, which are not in the rank-pair set of the key (the same as the current key), are moved into its rank-pair set. The procedure for calculating all possible keys and their associated rank pairs is not outlined in detail due to space constrains.

In constructing the 2-D-S tree, it is unnecessary to consider both of the keys $(A = B)$ and $(B = A)$ or $(A : B)$ and $(B : A)$ since they are equivalent. In general, the key is in the form of $A\theta B$, where A and B are symbols in V and $\theta \in \{<, =, :\}$. If θ is '=' or ':', $A \leq B$ (in alphabetical order). Consider the following 2-D strings:

$\{1, (C < A < B,\ C < A = B)\}$
$\{2, (C < B,\ C < B)\}$
$\{3, (A < B < B,\ B < B = A)\}$
$\{4, (A < B = C < A,\ A < A < C < B)\}$

All the possible keys are $\{A < A, A < B, A = B, A < C, B < A, B < B, B < C, C < A, C < B\}$. The leaf nodes for these keys, as discussed in Sect. 3.1, and the full tree constructed for all possible keys are given in Figure 3. Note that the maximum number of all possible keys is the function of different symbols. The number of possible keys is not very large even where the number of symbols are large. We discuss it in detail in Sect. 4.1.

3.3. Search

Before we discuss the search algorithm, let's use some examples to show what a 2-D-S-tree would look like and how it works. Consider the example tree given in Figure 3. Suppose a $type-1$ query is $(C < B,\ C < B)$. We first use $C < B$ as the key to traverse the 2-D-S tree and find the leaf node to hold the record with $C < B$ as its key. Then we fetch all the *id*s in *x_set*. Next, we similarly use $C < B$ as the key to fetch all the *id*s in its *y_set*. The results should then be $x\_set \cap y\_set$, and they are $\{1, 2\}$. Thus $\{1, (C < A < B,\ C < A = B)\}$ and $\{2, (C < B,\ C < B)\}$ are the results for the query.

In the next example, we use the same query but this time the type of the query is $type-2$. We can use the same procedure as above to get the result set $\{1,2\}$. However, we need to check their *rank* sets as well. We find that the only element in the *rank* set of $id = 1$ from *x_set* is $[1,3]$. Since $3-1=2$ is not equal to the $r(B)-r(C)=1$ (in the query), $id = 1$ should be eliminated from the result set. Further checking confirms that $id = 2$ meets the condition of $type-2$, thus the final result for the query is $\{2,(C<B,\ C<B)\}$.

If we have $type-1$ query $(C<A<B,\ C<B=A)$, then we need to use $C<A$ and $A<B$ as the keys to traverse the 2-D-S tree and get the intersection of their corresponding *x_set* parts, that is, $\{1,4\}\cap\{1,3,4\}=\{1,4\}$. But here we have to make sure that in the 2-D strings with $id = 1$ and $id = 4$, $C<A$ is ahead of $A<B$ in the string. This can be done by checking their *rank* pairs: $[1,2]$ for $id = 1$ in the record of $C<A$, and $[2,3]$ for $id = 1$ in the record of $A<B$. Thus, we know that in the first 2-D string $C<A$ is ahead of $A<B$. However, the fourth 2-D string doesn't meet the condition, since $[2,3]$ for $id = 4$ is in the record of $C<A$, and $[1,2]$ for $id = 4$ in the record of $A<B$ (here $3>1$). Thus, $id = 4$ should be eliminated from the result set. Second, in the same way, we search the tree by using $C<B$ and $A=B$ (notice that $A=B$ and $B=A$ are equivalent) as the keys from their *y_set* parts, and get the resultant set, which is also $\{1\}$. Therefore, the final result for the query is $\{1,$ $(C<A<B,\ C<A=B)\}$.

Finally, if we have a $type-0$ query, $(C<A<B,\ C<B=A)$, for the *x_set* part, the searching procedure is the same as in the previous example. For the *y_set* part, however, according to the definition of the $type-0$ query, we know that both $B:A$ and $B<A$ are $type-0$ subsequences of $B=A$, thus we should use $C<B$, $B=A$, $B:A$ and $B<A$ as the keys to search the tree. But since the records in the leaf nodes are ordered by their keys, and all the leaf nodes are linked by using the "next page" pointers, we are able to conduct a range search for $B=A$, $B:A$ and $B<A$. Thus, we need to search the tree from the root only once for these three keys. Again final result for the query is $\{1,(C<A<B,\ C<A=B)\}$.

In general, the search algorithm first traverses the tree using pairs of consecutive symbols in the query 2-D string as the keys to the leaf node, where all their rank pairs reside. It then calculates rank pairs to determine the results for different types ($type-0$, $type-1$, and $type-2$) of queries. Again, the search algorithm is not given in detail due to space constrains.

3.4. Discussion

It might be tempting to build inverted lists for the records instead of building a tree-structured index. This approach, however, has a number of problems. First of all, we need to conduct range searches for $type-0$ queries. This requires that the records for the key immediate following the current key can be accessed directly without going through the index again. It is well-known that inverted files do not support range queries efficiently. Second, all the records for a specific key need to be physically grouped together to support efficient searches. These are not guaranteed in an inverted file. Finally, for the database sizes we are considering, the inverted file size would be too large to keep in memory. The size of the file is sensitive to the number of images in the database. Maintenance of inverted file on secondary storage and performing searches on it are problematic.

4. PERFORMANCE ANALYSIS

We implemented the 2-D-S-tree and the sequential 2-D string matching algorithm (labeled as "S-Match" in the tables given in this section) using the EXODUS[24] storage manager $V3.0$. The experimental environment consists of two Sun SPARCstation 5's in a client-server configuration. The implementation was done in C^{++} (eg^{++}). We conducted a number of experiments to study the behavior of the algorithms. Table 1 lists the parameters used in the cost analysis and the values assigned to those parameters used in the experimental evaluation. The default values for D, S and L are set to be 5000, 40 and 10, respectively. We generated the symbols S in all the experiments according to *normal* distribution. This means that some of the symbols would appear in more 2-D strings than others to reflect the reality that some of the objects do appear in more images than others.

We also maintain a list to organize all 2-D strings in the database. The list is used for the sequential 2-D string subsequence matching and also for fetching 2-D strings for a given query after the 2-D-S tree is searched (we only keep the *id*s of 2-D strings in the leaf nodes of the tree). The list contains record entries of the form $\{id$, 2-D *string*, *pointer*$\}$, where *id* is a unique identifier for each 2-D string (a monotonically increasing number assigned to the 2-D string as it is entered into the database), and the *pointer* is the physical address of the original image in database. The list is ordered according to *id*s, therefore, once we know all the *id*s of 2-D strings for a query, we can retrieve the resulting images from the database by searching this list. Of course, the list can be organized using hashing techniques to speed up searches of it.

It would have been preferable to conduct the experiments on real image repositories. Unfortunately, most of the publicly available image repositories contain images of single objects (e.g., cars, animals, flowers). For the types of queries we are interested in, images should have a number of objects. These types of repositories are usually maintained by news organizations as photo archives and are not generally publicly available. We are now in the process of setting up a repository of these images which we will use in future studies.

Table 1. Parameters and notation

Parameters	*Meaning*	*Values*
D	number of images (2-D strings) in database	2000, 5000, 10000, 15000
S	number of different symbols in V ($\mid V \mid$)	20, 40, 60, 200, 500
L	maximum number of symbols in a 2-D string	7, 10, 15
$sz(N)$	node size (page size)	$4K$ bytes
$sz(P)$	number of bytes for a node pointer	16 bytes
$sz(K)$	number of bytes for the key	20 bytes
$sz(Ctr)$	number of bytes for the counter	4 bytes
$sz(Cid)$	number of bytes for the 2-D string *id*	4 bytes
$sz(Off)$	number of bytes for specifying the offset	4 bytes

4.1. Database Sizes

In the first set of experiments, we studied the effect of the database size (i.e., number of images) on the query performance. We fixed the values of S $(= 40)$ and L $(= 10)$ and randomly generated from 2000 to 15000 2-D strings. Table 2 shows that the sequential matching degrades quickly as the database size grows. This is because, for a larger database, the query 2-D string has to match against more 2-D strings (sequentially). However, the performance of the 2-D-S-tree is more or less constant. This can be explained as follows. As mentioned before, in the 2-D string technique, a query can be represented by a 2-D string as well. Let L_q be the number of symbols appearing in the query 2-D string. For example, for a query: $(B < A < B,\ B < B = A)$, its $L_q = 3$. Let H be the height of the 2-D-S-tree. Then, the 2-D-S-tree query costs in general are

$$Q_c = (L_q - 1) * (H + op) + O((L_q - 2) * M) \tag{1}$$

where op is the average number of overflow pages for the records, and where M is the average number of ids in the records for a 2-D-S-tree. The number of leaf nodes entries E is

$$E = S * S + 2 * (S + (S - 1) + \cdots + 1)) = 2 * S^2 + S \tag{2}$$

For example, if $V = \{A, B\}$, then $S =\mid V \mid= 2$ and $E = 2 * 2^2 + 2 = 10$. In this case, the keys are $\{A < A,\ A < B,\ B < A,\ B < B,\ A = A,\ A = B,\ B = B,\ A : A,\ A : B,\ B : B\}$. As mentioned before, non-leaf nodes in a 2-D-S-tree are similar to those in a B$^+$ tree. We should keep in mind that a fanout (the number of keys or separators

Table 2. Effect of database size (seconds)

Database Size	$type - 0$		$type - 1$		$type - 2$	
(no. of images)	S-Match	2-D-S-tree	S-Match	2-D-S-tree	S-Match	2-D-S-tree
2000	167.400	0.750	104.900	0.080	79.000	0.070
5000	350.200	0.790	279.510	0.120	202.990	0.120
10000	$---$	1.020	569.160	0.120	364.170	0.110
15000	$---$	1.100	927.020	0.200	629.190	0.170

Table 3. Effect of vocabulary size (seconds)

Vocabulary Size	*type-0*		*type-1*		*type-2*	
(no. of symbols)	S-Match	2-D-S-tree	S-Match	2-D-S-tree	S-Match	2-D-S-tree
20	322.314	0.920	247.962	0.207	166.591	0.180
40	350.200	0.790	279.510	0.120	202.990	0.120
60	363.170	0.570	291.230	0.090	207.920	0.090
200	398.670	0.560	357.230	0.090	234.250	0.082
500	407.840	0.530	369.130	0.087	254.190	0.080

in the non-leaf node) of 100 for B^+ trees is not unusual. With two levels, we can access up to 10,000 records, and with three levels up to 1,000,000 records. B^+ trees are very broad, rather than high. A typical B^+ tree has only 1 to 3 levels. For a 2-D-S-tree, even if $S = 5000$, the total leaf nodes entries are $2 * 500^2 + 500 = 500,500$. Therefore, the 2-D-S-tree is very broad as well, that is to say, most likely $H \leq 3$.

The important factors in the query cost, then, are L_q and M (op is determined by the number of ids in the records). The maximum number of possible symbol pairs (in the form of $A\theta B$, $\theta \in \{<, =, :\}$) in a 2-D string with length of L is

$$K = \frac{L * (L-1)}{2} \tag{3}$$

Then the average number of ids in a record for a 2-D-S-tree, M, can be estimated as

$$M = \frac{K}{E} * D = \frac{L * (L-1)}{4 * S^2 + 2S} * D \tag{4}$$

For example, if $L = 10$ and $S = 40$, when $D = 2000$, $M \approx 28$, and when $D = 15,000$, $M \approx 208$. If the page size is $4K$ bytes, and the id and the $rank$ size are 4 bytes (they are integers), even in the case of $D = 15,000$, it is likely that most records are small and can be fitted in one page, so the overflow pages are not needed for most leaf nodes. In general, records will grow large to require overflow pages only when S is very small (i.e., when there are very few different symbols in the database) and the database size is very large.

From the above analysis, we know that the query cost for a 2-D-S-tree is approximately equal to (a small number of) page access cost, plus time spent on checking resultant ids to see if they satisfy the conditions for $type - i$ ($i = 0, 1, 2$) queries. This is polynomial ($(L_q - 2) * M$ in equation 1). Thus we can expect that searches over the 2-D-S-tree to be very efficient.

4.2. Vocabulary Sizes

In the second set of experiments, we fixed the values of D (= 5000) and L (= 10) and varied S from 20 to 500. Table 3 shows that the effect of the number of different symbols, on both the sequential matching and the 2-D-S-tree, is not very significant. The reasons that the performance of the 2-D-S-tree improves slightly as the vocabulary size grows are as follows. First, as analyzed above, the increase of the vocabulary size does not raise the height of the tree significantly (actually, for these experiments, the height of the tree remained as 2). Second, from equation 4, we know that as the vocabulary size S increases, the average number of ids in a record M decreases. Therefore, the time spent on checking resultant ids is reduced. In general, the increase of the vocabulary size favors the 2-D-S-tree (as shown in equation 4). This is really important for some applications, such as news image databases, in which the vocabulary size can be very large. For example, a query like "Find all images of Clinton and Dole in front of the White house" needs to have words (or symbols) for Clinton, Dole and White house in the vocabulary set instead of only using more general terms like person or house.

4.3. 2-D String Length

In the third set of experiments, we fixed the values of D (= 5000) and S (= 40) and varied L from 7 to 15. Table 4 shows that the sequential matching is so sensitive to the 2-D string length that it degrades very quickly as L grows, since L is an important factor for the efficiency of the sequential matching. The performance of the 2-D-S-tree

Table 4. Effect of 2-D string length (seconds)

Vocabulary Size (no. of symbols)	*type-0* S-Match	*type-0* 2-D-S-tree	*type-1* S-Match	*type-1* 2-D-S-tree	*type-2* S-Match	*type-2* 2-D-S-tree
7	43.200	0.590	22.100	0.100	21.710	0.080
10	350.200	0.790	279.510	0.120	202.990	0.120
15	$---$	0.92	$---$	0.200	334.840	0.170

degrades gradually as L grows. From equations $1-4$ and the above analysis, we know that when $D = 5000$ and $S = 40$, and L is assigned to 7, 10 and 15, respectively, the height of the tree, H, and the average number of overflow page(s), *op*, will not change, but the average number of *ids* in the records M changes to 9, 13, and 21, respectively. This increases the time $O((L_q - 2) * M)$ and, as a result, increases the query costs.

We should point out that the 2-D string technique was primarily designed for iconic indexing,[8] not for exactly depicting the original images. From this perspective, for a given image, if a 2-D string of the image can provide sufficient discrimination from other images in the database, it is considered "good." In other words, given the fuzziness of multimedia queries, we are generally satisfied even if the system generates some false hits. For example, for the image in Figure 1(a), the 2-D string $(tree < house : dog = sun < plane,\ house : dog < tree < sun = plane)$ is satisfactory, since it presents the most important spatial relationships among objects in the image, and provides sufficient information to discriminate the image from other images. Notice that we ignored, in the 2-D string representation, some objects and their spatial relationships with others in the image, such as the *window* and the *door* on the wall of the house, the *grass* on the ground, and the *bird* on the roof of the house, since they are not as important as the others from indexing point of view. In general, if the vocabulary size is S, and the 2-D string length is L, then $(3^{L-1} * S^L)^2$ 2-D strings can be composed, which, in turn, can represent $(3^{L-1} * S^L)^2$ different images. For example, if a 2-D string length is 5, the 2-D string can discriminate itself from $(3*S^2)^2+(3^2*S^3)^2+(3^3*S^4)^2+(3^4*S^5)^2$ 2-D strings. Therefore, the 2-D string length doesn't need to be very long when using 2-D-S-tree for indexing, if S is not too small (say less than 10).

4.4. Query 2-D String Length

The number of symbols appearing in the query 2-D string L_q is an important factor for the efficiency of the 2-D-S-tree. From equation 1, we know that it affects the number of pages that need to be fetched, and the time spent on checking resultant *ids*. In this experiment, we arbitrarily limited L_q to a certain number, 4, to further enhance the search efficiency. For example, if the query 2-D string is:

$$(R < A < T < Y < J = G = W = D = V = R,\ W < Y = V < R < H = A = J < R < D < T),$$

we cut it to the shorter string $(R < A < T < Y,\ W < Y = V < R)$ to conduct the search.

In this set of experiments, for each experiment, we randomly generated 100 query 2-D strings with their length larger than 4, then we used their short forms ($L_q = 4$) to conduct searches. Table 5 shows the results of the experiments (number of false hits). The results indicate that the query 2-D string length, L_q, can be short as long as S is sufficiently large.

This experiment was conducted to check whether the search performance can be improved by shortening the search string. This makes sense because of the following reasons. First, the basic requirement of indexing in the image database is that it provide sufficient discrimination to prevent the retrieval of a large fraction of the database, and not that it produce only the exact match. Thus, partial queries are common. Second, the main distinction between pattern matching and searching in the image database is that while exact pattern matching is the main criterion in pattern recognition, in database retrieval the objective is to help the user navigate through a large number of images. In fact, the users may not even be interested in the best match at all. They might use the retrieved patterns to further modify the search specifications (incremental queries). Thus, the constraints on pattern recognition are somewhat relaxed in the database environment, and efficiency is related to reducing the retrieval time, and robustness in terms of not missing targets (allowing some *false hits*, but no *false dismissals*) is a larger concern. Notice that "false hits" are acceptable, since they can be discarded easily through a post-processing step.

Table 5. False hits

Database Size	*Vocabulary Size*	*2-D String Length*	$type - 0$	$ype - 1$	$type - 1$
5000	40	10	0	0	0
10000	40	10	0	0	0
15000	40	10	1	0	0
5000	20	10	2	1	0
5000	60	10	0	0	0
5000	40	7	0	0	0
5000	40	15	1	0	0

Table 6. Space requirements (number of pages)

Database Size	*2-D-S-tree*
2000	454
5000	935
10000	1403
15000	2105

THEOREM 4.1. *There are no false dismissals when we use the short form of the query 2-D string to conduct the search on behalf of the original query 2-D string in a 2-D-S-tree.*

Proof. (*Sketch*) A false dismissal of an image (a 2-D string) by a query means that the image (the 2-D string) is not returned when it is a resultant image (2-D string) of the query. In the 2-D-S-tree, if a 2-D string is "hit" by a query, it must be "hit" by the short form of the query. Thus, there are no false dismissals in the 2-D-S-tree. For example, if a 2-D string contains a query 2-D string ($R < A < T < Y < J = G = W = D = V = R,\ W < Y = V < R < H = A = J < R < D < T$) as a $type - 1$ or $type - 2$ subsequence, it must contain the short form of the query 2-D string ($R < A < T < Y,\ W < Y = V < R$) as a $type - 0$ or $type - 1$ or $type - 2$ subsequence as well. □

Finally, the possibility of false hits is very small if S is sufficiently large (say larger than 10), since the hits themselves are small. That is to say, using the above example, the probability of a 2-D string containing both $R < A < T < Y$ in x-axis and $W < Y = V < R$ in y-axis is small. The probability can be roughly estimated as (if $L_q = 4$)

$$P \approx \frac{(L-3)^4}{3^3 * S^4} * \frac{(L-3)^4}{3^3 * S^4} = 1/729 \left(\frac{L-3}{S}\right)^8 \qquad L \geq 3 \tag{5}$$

If $S > L$, P is very small.

Table 7. Space Requirements (number of pages)

Vocabulary Size	*2-D-S-tree*
20	913
40	935
60	936
200	1294
500	3376

4.5. Storage Requirements and Update Costs

In the storage requirement study, we varied the database size (D), the symbol set size (S), and the maximum length of 2-D strings (L) with the default values fixed as before. The page size in this set of experiments is fixed at $4K$. Tables 6, 7 and 8 show that the required storage space increases as D or L grows, due to an increase in the number

Table 8. Space Requirements (number of pages)

2-D String Length	*2-D-S-tree*
7	488
10	935
15	2005

Table 9. Update Costs (seconds)

2-D String Length	*Insertion*	*Deletion*
7	0.4400	0.0500
10	0.5200	0.1300
15	2.3000	1.0200

of *id*s in the leaf nodes. But the vocabulary set size does not have much effect on the storage requirement of the 2-D-S-tree. We should point out that in the case of 2000 $4K$ pages, the 2-D-S-tree needs about 8 Mbytes in some cases (in Table 6 and Table 8). This is quite reasonable, especially if one considers that this is the space requirement for only about 10 raw images. Table 9 shows the results of the experiments for update costs. The performance of both insertions and deletions degrades as the 2-D string length grows, due to the increase in the number of potential keys in 2-D strings which were inserted into or deleted from the 2-D-S-tree.

5. CONCLUSIONS

Spatial features represent the spatial relationships among objects in an image, such as directional relationships, adjacency, overlap, and containment involving two or more objects. These are very important for content-based retrieval of images in a image database. 2-D string is the most common data structure that is used to facilitate this type of retrieval. However, searches over a large set of images is done sequentially, making the technique inefficient for large databases. In this paper, we propose a new indexing structure, called the 2-D-S-tree, to organize 2-D strings for query efficiency. The 2-D-S-tree is completely dynamic; inserts and deletes can be intermixed with searches and no periodic reorganization is required. A performance analysis is conducted, and both analytical analysis and experimental results indicate that the 2-D-S-tree is an efficient index structure for content-based retrieval of images.

We should point out that any index structure, including hashing,[25,26] B^+ trees and R^*-trees, will suffer from a very skewed data distribution and the 2-D-S-tree is no exception. For example, in a facial image database, there are a very few symbols such as *mouth, eye, nose* and *ear* etc., and most symbols will appear in all the images, and most spatial relationships among objects are almost the same. From the performance analysis in previous section, we know that the 2-D-S-tree may not work well for these cases. Generally, 2-D-S-tree and 2-D string scheme work well for a database where there are a variety of images with different objects and spatial relationships among objects, such as a news image database which we are currently focusing on.

In the image DBMS that is under development, we identify the most interesting objects, called salient objects, as well as the spatial relationships among these salient objects, and then use 2-D-S-tree to index them. Our experience justifies the claim we made in the performance analysis that the 2-D string length for each image doesn't need to be long, because, only the identified salient objects are likely to be queried even if there might be more objects in an image. For these cases, the 2-D-S-tree index works very well.

We are currently working on extensions to 2-D-S string indexing. The current scheme identifies only three types of relationships among salient objects: *to-the-left-of* (or *alone*), *in-the-same-quadrant* and *aligned-the-same* along one of the two axes. However, the set of topological and directional relations that might exist among salient objects are much richer (e.g., *overlaps, contained-in*). We are working on extending the 2-D string mechanism to deal with these types of relationships.

ACKNOWLEDGMENTS

This research has been supported by a grant from the Canadian Institute for Telecommunication Research (CITR), a Federal Network of Centre of Excellence funded by the Government of Canada and by a Strategic Grant from the Natural Science and Engineering Research Council (NSERC) of Canada.

REFERENCES

1. W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, and P. Yanker, "The QBIC project: Querying images by content using color, texture, and shape," *Proc. 1st SPIE Conf. Storage and Retrieval for Image and Video Databases* , pp. 173–187, 1993.

2. E. Binaghi, I. Gagliardi, and R. Schettini, “Indexing and fuzzy logic-based retrieval of color images,” *Proc. 2nd Int. Conf. Visual Database Systems* , pp. 79–92, 1992.
3. T. Hermes, C. Klauck, and J. Zhang, “Image retrieval for information systems,” *Proc. 3rd SPIE Conf. Storage and Retrieval for Image and Video Databases* , pp. 394–405, 1995.
4. S. K. Chang, C. W. Yan, D. C. Dimitroff, and T. Arndt, “An intelligent image database system,” *IEEE Trans. Software Eng.* **14**(5), pp. 681–688, 1988.
5. H. Jagadish, “A retrieval technique for similar shapes,” *Proc. ACM SIGMOD Int. Conf. on the Management of Data* , pp. 208–217, 1991.
6. C. Faloutsos, *Searching Multimedia Databases by Content*, Kluwer, Boston, 1996.
7. J. R. Bach, C. Fuller, A. Gupta, A. Hampapur, B. Horowitz, R. Humphrey, R. Jain, and C.-F. Shu, “The Virage image search engine: An open framework for image management,” vol. Proc. 4th SPIE Conf. Storage and Retrieval for Still Image and Video Databases, pp. 76–87, 1996.
8. S. K. Chang, Q. Y. Shi, and C. W. Yan, “Iconic indexing by 2D strings,” *IEEE Trans. Pattern Anal. and Machine Intel.* **9**(3), pp. 413–428, 1987.
9. C. C. Chane and S. Y. Lee, “Retrieval of similar pictures on pictorial databases,” *Pattern Recognition* **24**(7), pp. 675–680, 1991.
10. S. K. Chang, E. Jungert, and Y. Li, “Representation and retrieval of symbolic pictures using generalized 2D strings,” *Proc. SPIE Conf. Visual Commun. and Image Processing* , pp. 1360–1372, 1989.
11. S. Y. Lee and F. J. Hsu, “Retrieval of similar pictures on pictorial databases,” *Pattern Recognition* **23**(10), pp. 1027–1034, 1990.
12. C. C. Chane and S. Y. Lee, “Relative coordinates-oriented symbolic string for spatial relationship retrieval,” *Pattern Recognition* **28**(4), pp. 563–570, 1995.
13. S. Y. Lee, M. C. Yang, and J. W. Chen, “2D B-string: A spatial knowledge representation for image database systems,” *Proc. Second Int. Computer Sci. Conf.* , pp. 609–615, 1992.
14. N. Beckmann, H. Keiegel, R. Schneider, and B. Seeger, “The R*-tree: An efficient and robust access method for points an d rectangles,” *Proc. ACM SIGMOD Int. Conf. on the Management of Data* , pp. 322–331, 1990.
15. A. Guttman, “R trees: A dynamic index structure for spatial searching,” *Proc. ACM SIGMOD Int. Conf. on the Management of Data* , pp. 47–57, 1984.
16. T. Sellis, N. Roussopoulos, and C. Faloutsos, “The R^+ tree: A dynamic index for multidimensional objects,” *Proc. Int. Conf. on Very Large Data Bases* , pp. 507–518, 1987.
17. N. Roussopoulos and D. Leifker, “Direct spatial search on pictorial database using packed R-trees,” *Proc. ACM SIGMOD Int. Conf. on the Management of Data* , pp. 17–31, 1985.
18. I. Gargantini, “An effective way to represent quadtrees,” *Comm. ACM* **25**(12), pp. 905–910, 1982.
19. J. Nievergelt, H. Hinterberger, and K. C. Sevcik, “The grid file: An adaptable symmetric multikey file structure,” *ACM Trans. Database Syst.* **9**(1), pp. 38–71, 1984.
20. J. T. Robinson, “The K-D-B-Tree: A search structure for large multidimensional dynamic indexes.,” *Proc. ACM SIGMOD Int. Conf. on the Management of Data* , pp. 10–18, 1981.
21. C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, “Fast subsequence matching in time-series databases,” *Proc. ACM SIGMOD Int. Conf. on the Management of Data* , pp. 419–429, 1994.
22. V. Oria, M. T. Özsu, L. Liu, X. Li, J. Li, Y. Niu, and P. Iglinski, “Modeling images for content-based queries: The DISIMA approach,” *Proc. 2nd Int. Conf. on Visual Information Systems* , pp. 339–346, 1997.
23. D. Comer, “The ubiquitous B-tree,” *ACM Comp. Surv.* **11**(2), pp. 121–137, 1979.
24. M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita, “Object and file management in the EXODUS extensible database system,” *Proc. Int. Conf. on Very Large Data Bases* , pp. 91–100, 1986.
25. W. Litwin, “Linear hashing: A new tool for file and table addressing,” *Proc. Int. Conf. on Very Large Data Bases* , pp. 212–223, 1980.
26. R. Fagin, J. Nievergelt, N. Pippenger, and R. Strong, “Extendible hashing: A fast access method for dynamic files,” *ACM Trans. Database Syst.* **4**(3), pp. 315–344, 1979.