Abstract

An efficient index structure for complex spatial objects is one of the most challenging requirements in non-traditional applications such as geographic information systems (GISs), computer-aided design (CAD), and multimedia databases. In this paper we first propose an extension of an existing index structure called the two-step index structure (TSIS). The TSIS integrates two index structures, one for original objects and the other for their decomposed components. Then, we present a cost model that predicts the performance of the TSIS. In contrast to several earlier investigations on this subject which only considered the filter step, we take into account the performance of the refinement step. Experimental results show that the cost model is accurate, the relative error being below 15%. The performance of our index structure is compared with that of a state-of-the-art index structure by experimental measurements. Our index structure outperforms the state-of-the-art index structure due to its ability to reduce a large amount of storage. © 2000 Published by Elsevier Science Inc. All rights reserved.

Keywords: Two-step index structure; Complex spatial objects; Object decomposition method; Cost model; Performance of refinement step

* Corresponding author.
E-mail addresses: yongju@sangju.ac.kr (Y.-J. Lee), chungcw@islab.kaist.ac.kr (C.-W. Chung).
1. Introduction

It has been recognized that existing conventional index structures are not suitable for non-traditional complex applications such as geographic information systems (GISs), computer-aided design (CAD), and multimedia databases. Classical one-dimensional index structures such as the $B^*$-tree and hash tables are designed to deal with simple data types such as integer, float, string, and time. However, the majority of non-traditional applications require the management of spatial objects (i.e., objects related to the space). Spatial objects may be composed of geographic data such as irregular-shaped maps, or multi-dimensional data such as contours of CAD objects. Complex properties of such spatial objects would be difficult to predict and to manipulate. Therefore, one of the most challenging requirements is to develop an efficient index structure for complex spatial objects.

Two well-known methods, the minimum bounding rectangle (MBR) and the region decomposition, can be used for indexing spatial objects. The MBR, the smallest aligned rectangle enclosing an object, enables a fast execution of spatial queries since testing MBRs against the query condition is much faster than testing exact objects. Some of the spatial index structures based on the MBR are the LSD-tree [1], the $R^+$-tree [2], and the X-tree [3]. The region decomposition uses the principle of ‘divide and conquer’ techniques. The data space is recursively divided into sub-regions until a region obtains a desired simple component. The Quad-tree [4], the Z-ordering [5], and the TR*-tree [6] fall under this method.

Spatial query processing based on the MBR is typically executed in two steps [7]. The first step, called the filter step, reduces the entire set of objects to a subset of candidates using their MBRs. This filter step is based on spatial index structures managing MBRs. The filter step does not exactly evaluate the query since the simple rectangle cannot exactly represent an irregularly shaped spatial object. Therefore, the candidate objects (CO) have to be examined in the subsequent step. The second step, called the refinement step, inspects the exact representation of each object of the candidates. This step usually applies complex algorithms known from the field of computational geometry to the COs.

At first glance, the MBR method seems to be a good approach for spatial query processing. However, a more detailed consideration reveals two major disadvantages:

1. Due to a rough approximation of the MBR, candidates may contain a number of ‘false hits’ not fulfilling the query condition. Moreover, the whole set of candidates has to be transmitted to the refinement step, even if they result in false hits.

2. The refinement of an object is costly if the object is complex. For some applications, the refinement step may not be a time consuming work because the CPU is fast. For many GIS applications, however, it is not uncommon...
to deal with tens of thousands or even hundreds of thousands of vertices. The brute-force \(O(n)\) algorithm is obviously inadequate for such applications.

In contrast, the region decomposition method, which decomposes a complex spatial object into a number of simple spatial components such as quadrants, trapezoids, and line segments, leads to both a better approximation quality and simpler spatial objects. However, since the number of components depends on the shape of the object, the decomposition process over the complex spatial object typically generates a large number of components each of which occupies a very small area. A number of decomposed components could result in a storage and query processing overhead.

To solve the problem of the region decomposition, we proposed a new decomposition method called the \textit{DMBR (Decomposed MBR)} [8]. The basic idea of this method is that an object is recursively divided into two components by splitting its MBR until a given constraint is satisfied. This enables a natural trade-off between the number and the complexity of decomposed components, as the number of components can be controlled by the given constraint. The success of this method depends on the ability to narrow down quickly a set of components that are affected by spatial queries. In order to decide which components are relevant for a particular geometric test, we need an efficient index structure that can organize the set of components. However, the main subject of [8] was not to discuss which index structure is the most suitable for organizing decomposed components but to develop the new decomposition method.

In this paper, we first propose a novel index structure based on the DMBR method. The proposed index structure consists of two levels. On the first level a spatial index structure (\(R\)-tree) is used for organizing MBRs of objects. Attached to the MBR is a reference to the exact representation of the object. The second level consists of a set of two-dimensional binary trees, called here \textit{DR-trees (Decomposed Rectangle trees)}, which are used for organizing decomposed components. Then, we present a cost model that predicts the performance of the proposed index structure. In contrast to several earlier investigations on this subject, we take into account the performance of the refinement step as well as the filter step. We derive analytical formulas to evaluate the average response time for point queries, region queries, and spatial join queries. Experimental results show that the cost model is accurate, the relative error being below 15\%. The performance of our index structure is compared with that of a state-of-the-art index structure by experimental measurements. This comparison shows that our index structure is superior to the state-of-the-art index structure.

This paper is organized as follows. Section 2 provides a summary of the DMBR method. Section 3 describes an extension of an existing spatial index structure called the two-step index structure (TSIS). Section 4 presents a cost
model that predicts the performance of the TSIS. Section 5 contains the performance evaluation. Conclusions are made in Section 6.

2. Overview of DMBR method

In [8] we proposed a new object decomposition method called the DMBR method. The basic idea is to divide a complex object into two components corresponding to disjoint half regions of its MBR space. Then a new MBR (called here a DMBR) for each of those components is generated. This operation is performed recursively along the vertical boundary and the horizontal boundary in strictly alternating sequence until every DMBR fulfills a given constraint. The constraint is expressed by the accuracy of the decomposition (AOD). This means that a split is permitted if the size of the resulting DMBR is above a threshold. The threshold is controlled by parameter $g$. That is, AOD($g$) requires a split of the DMBR that covers more than $2^{-g}$ of the MBR space.

The procedure on how to decompose a given spatial object is illustrated by means of an example. Consider a polygon shown in Fig. 1(a). This figure shows an MBR enclosing a spatial object. Assume that the threshold size is 25% of the MBR space, i.e., AOD(2). We subdivide the polygon until the given constraint is satisfied. At first, the polygon is divided into two sub-polygons by the vertical boundary. Then DMBRs for the sub-polygons are generated (see Fig. 1(b)). While the DMBR of the left sub-polygon is less than $2^{-2}$ of the MBR space, the DMBR of the right sub-polygon is larger than $2^{-2}$ of the MBR space. Thus, the object decomposition on the right sub-polygon is performed recursively along the horizontal boundary (see Fig. 1(c)). The recursive decomposition then terminates because every DMBR covers less than $2^{-2}$ of the MBR space.

The DMBR method has a parameter $g$ that controls the amount of redundancy for each object. At a low value of the parameter, the number of components can be minimized, but this decomposition provides a rather rough approximation of the object. On the other hand, the accuracy of the approx-
imation can be better at a higher value, but a linear increase in the number of components can be observed. From this observation, we can conclude that there is a balanced ratio between the number of components and the accuracy of the approximation. From extensive experiments in [8], an optimal value of the parameter can be obtained at around $g = 3$.

3. Two-step index structure

The success of the decomposition approach depends on the ability to narrow down quickly a set of components that are inspected by spatial queries. In order to decide which components are relevant for a particular geometric test, we need an efficient index structure that can organize the set of components. Although a number of spatial index structures based on the MBR have been developed, these structures are considered unsuitable for organizing the decomposed components, because components with the same identifier are randomly distributed on secondary storage. Arbitrary distribution of these objects over secondary storage leads to high access cost during query processing.

We propose an extension of the existing index structure called the TSIS, which integrates two index structures for original objects and their decomposed components. Among existing index structures, the $R^*$-tree is selected to index original objects, since popular storage systems such as the SHORE [9] already use the $R^*$-tree as their basic index structure. Obviously, instead of the $R^*$-tree, any other spatial index structures such as the LSD-tree and the X-tree might be considered for designing our index structure. Since the cost of implementing a new index structure can be more expensive than the cost of extending an already existing one, adding new features to the existing index structure is an excellent alternative.

Fig. 2 depicts our TSIS schematically. Two parts are distinguished: (1) the first level is a spatial index structure for original objects, which is a straightforward modification of the $R^*$-tree; and (2) the second level is a set of two-dimensional binary trees, called here DR-trees, which are designed to store decomposed components and to handle a set of decomposed components in main memory. The $R^*$-tree consists of leaf and non-leaf nodes. MBRs of original objects are stored in the leaf nodes. Each leaf node is supplemented by a pointer to the DR-tree and by an object identifier of the original object. A non-leaf node is built by grouping rectangles at the lower level. In the DR-tree, DMBRs and their component identifiers are stored at leaf nodes, and rectangles enclosing components are stored at non-leaf nodes.

The structure of the DR-tree is similar to that of the LSD-tree [1]. However, the distinctive feature of the DR-tree is that it specifies a region by the DMBR. A non-leaf node of the DR-tree contains an entry of the form $(\text{left-ptr, Rectangle, right-ptr})$, where left-ptr and right-ptr are pointers to the left and right
child node of the tree and *Rectangle* is a representation of the enclosing rectangle. A leaf node contains an entry of the form \((\text{Cid}, \text{Rectangle})\), where \text{Cid} is a component identifier that refers to the representation of a decomposed component and \text{Rectangle} represents the DMBR of a decomposed component. Considering the object decomposition illustrated in Fig. 1, for example, a set of the DMBRs and its corresponding DR-tree are shown in Fig. 3(a) and (b), respectively.

The DR-tree is persistently inserted into secondary storage and completely transferred into main memory when a complete object is required for a geometric operation. Therefore, it is not required to build up the DR-tree in main memory or to convert its pointers. The DR-tree is dynamic, i.e., insertions and deletions of objects can be intermixed with queries and no periodic global reorganization is required. The actual implementation of the DR-tree uses a
variable array. The array contains locations of the left and right child of non-leaf nodes, and points to the corresponding components of leaf nodes. This design has the following desirable properties: since each tree is stored as a separate record, we can take advantage of the fine-granularity (record-level) locking provided by a storage manager. It is easy to implement.

4. Analysis of two-step index structure

This section provides a cost model that estimates the performance of the TSIS. Powerful cost models are useful for three reasons [10]:
1. They allow us to better understand the behavior of an index structure under various input data sets with different sizes.
2. They can provide an objective comparison when different index structures are compared.
3. They can be used by a query optimizer in order to calculate the cost of a complex spatial query.

In the first subsection, we introduce a basic formula used throughout this section, and then continue in the other two subsections with the analysis of the filter step and the refinement step.

4.1. Cost model

Spatial query processing consists of the filter step and the refinement step. According to these two steps, the performance of the TSIS can be evaluated. The overall response time of spatial query processing is to add the response time of the filter step and the refinement step:

\[ T_{\text{query}} = T_{\text{filter}} + T_{\text{refine}} \]  

(1)

where \( T_{\text{query}} \) stands for the overall response time of a spatial query, \( T_{\text{filter}} \) for the response time of the filter step, and \( T_{\text{refine}} \) for the response time of the refinement step.

The performance of the filter step is determined by the performance of the index structure designed for MBRs. The response time of the filter step is mainly affected by the time required to retrieve the nodes touched by the query plus the time required by the CPU to process the nodes. Since the CPU is much faster than the disk, we assume that the CPU-time is negligible compared with the time required by a disk to retrieve pages. Thus the response time for the filter step is approximated by the number of nodes (pages) that will be retrieved by the spatial query:

\[ T_{\text{filter}} = \text{number of disk accesses} \cdot c_a \]  

(2)

where \( c_a \) is the disk access time.
The performance of the refinement step is determined by the time spent for the computational geometry algorithm. Specifically, COs produced by the filter step are fetched from a disk, and those objects are examined to determine whether they really satisfy the query condition. If we assume that every candidate object can be retrieved from a page, the response time of the refinement step can be expressed by:

\[ T_{\text{refine}} = \text{number of candidate objects}(c_a + \text{exact geometry test time}). \]  

(3)

We used the TSIS for spatial query processing. Under this structure, the R*-tree is used for the filter step. Before applying the refinement step, the DR-tree is used for pruning false hits quickly. For the performance evaluation, we include the time spent with the DR-tree in the time for the refinement step. This is due to the fact that the DMBR method simplifies the execution of complex computational geometry algorithms by using a set of decomposed components. Thus we assign the task of handling these decomposed components using the DR-tree to the refinement step.

4.2. Filter step analysis

This subsection describes the methodology, which was derived in [10,11], for evaluating the performance of the filter step. The performance is usually evaluated by the ability to answer region queries. Other queries such as point queries or spatial join queries are also important, but formulas for these queries can be easily obtained by extending that of region queries. Therefore, most efforts to analytically predict the performance of the filter step have concentrated on the region query (RQ) performance.

We introduce an analytical formula to evaluate the average response time for a query asking for all objects that intersect a query window \( q = (q_x, q_y) \). In this discussion we assume that queries are uniformly distributed over the unit square space, and \( N \) spatial objects are also uniformly distributed in the unit square space.

The next definition and lemma form the basis for the analysis.

**Definition 1.** The density \( D \) of a set of \( N \) spatial objects is the average number of spatial objects that contain a random point in the unit square space.

**Lemma 1.** If \( N \) spatial objects have MBRs with the average size \( n_x \times n_y \), then the density \( D \) is given by the following formula:

\[ D = N(n_x \times n_y). \]  

(4)
Proof. The probability that a random point falls in the rectangle \((n_x, n_y)\) is the area of the rectangle \(n_x \times n_y\). Therefore, the average number of spatial objects that contain the given point, i.e., density, is given by \(N(n_x \times n_y)\). □

Assume now an \(R^*\)-tree of height \(h\) (the root is assumed to be at level \(h\) and leaf nodes are assumed to be at level 1). The next lemma calculates the expected number of disk accesses (DA) for the RQ.

**Lemma 2.** If \(N_i\) is the number of nodes (i.e., MBRs) at level \(i\) of the \(R^*\)-tree and \(n_i = (n_{ix}, n_{iy})\) is their average node size, then the expected number of DA in order to answer an RQ \(q = (q_x, q_y)\) is defined as follows:

\[
DA = 1 + \sum_{i=1}^{h-1} \{N_i(n_{ix} + q_i)(n_{iy} + q_y)\}.
\]

(5)


The above lemma allows us to estimate DA for an RQ \(q\), but this lemma assumes that the \(R^*\)-tree has been built and that the size of each MBR of the \(R^*\)-tree can be measured. In [10] Lemma 2 was extended to express it without using \(n_i\). To do that, the paper showed that one only needed data properties \(N\) and \(D\), and the effective capacity \(C\) of the nodes of the \(R^*\)-tree as the average number of entries per node. Assume that the lengths of node sides are equal (i.e., \(n_{ix} = n_{iy} = n_i\)) and the sides of the query window are equal-sized (i.e., \(q_x = q_y = q\)), the following formula can be derived [10]:

\[
DA = 1 + \sum_{i=1}^{\left\lceil \log_2 \frac{C}{\sqrt{N/C}} \right\rceil} \left(\sqrt{D_i} + q\sqrt{\frac{N}{C}}\right)^2.
\]

(6)

From Eq. (6) we extract the following useful formula. If the query is a point query (PQ) \((q = 0)\), then

\[
DA = 1 + \sum_{i=1}^{\left\lceil \log_2 \frac{C}{\sqrt{N/C}} \right\rceil} D_i.
\]

(7)

If the query is a spatial join (SJ) query, we can view it as a series of region queries. Thus Eq. (6) can be repeatedly used in order to estimate the cost of the SJ query (see [12,13] for detailed explanations).

4.3. Refinement step analysis

Most of the work in the literature has dealt with the expected performance of the filter step. Our work extends the previous work for evaluating the per-
formance of the refinement step. The refinement step of an RQ is to inspect whether COs actually fulfill the query condition. The expected number of COs can also be computed using only the data properties \( N \) and \( D \).

**Theorem 1.** Given a set of \( N \) objects with average node size \( n \) and a query window with size \( q \), the expected number of CO is given by the formula:

\[
CO = \begin{cases} 
N\left(\sqrt{\frac{D}{N}} + \frac{q}{n}\right)^2 & \text{if } q < \sqrt{\frac{D}{N}}, \\
4D & \text{if } q \geq \sqrt{\frac{D}{N}}.
\end{cases}
\]  

(8)

**Proof.** In case \( q < n \) (i.e., \( q < \sqrt{\frac{D}{N}} \)), CO is equivalent to \( D \) if we ‘inflate’ the nodes of the R*-tree by \( q \) in the \( x \)-direction and \( y \)-direction. Using Lemma 1 we can express CO as

\[
CO = N(n + q)^2 = N\left(\sqrt{\frac{D}{N}} + q\right)^2.
\]

In case a query window is large compared with the average object area (i.e., \( q \geq \sqrt{\frac{D}{N}} \)), some objects can avoid the refinement step. For instance, when the MBR of an object is wholly contained within a query window, the refinement step is not needed. Thus COs requiring the refinement step are identified by excluding hits fulfilling the query condition from COs. Fig. 4 illustrates this idea by the dotted line. When \( q \geq \sqrt{\frac{D}{N}} \), CO is given by the average number of MBRs that contain corners of the query window. That is,

\[
CO = 4D.
\]

What remains is an estimation of the exact geometry test time. The simplest way to solve this problem is a sequential search: proceed around polygon

![Fig. 4. Hits fulfilling the query condition and a candidate object.](image-url)
vertices, edge by edge, testing each to see whether it intersects the query window. If an intersection is detected, the search stops successfully. Otherwise, the search runs until the last edge is processed. To detect an intersection, the average number of comparisons is about \( \frac{v}{2} \), where \( v \) is the average number of polygon vertices. This analysis is based on empirical evidences. Thus the exact geometry test time can be expressed by

\[
\text{exact geometry test time} = \frac{v}{2} c_c,
\]

where \( c_c \) is the comparison time.

Hence, from Eqs. (1)–(3), (6), (8) and (9) we have

\[
T_{\text{query}} = \left\{ 1 + \sum_{i=1}^{\left\lceil \log \frac{C}{N}\right\rceil} \left( \sqrt{D_i} + q \sqrt{\frac{N}{C}} \right)^2 + \text{CO} \right\} c_a + \text{CO} \frac{v}{2} c_c.
\]

From Theorem 1 we extract the following formula. If the query is a PQ (\( q = 0 \)), then

\[
\text{CO} = D.
\]

Since a PQ requires the evaluation of the complete shape of the spatial objects (i.e., ‘point-in-polygon’ test [14]), the number of comparisons for the PQ is clearly represented by \( v \). Thus the performance of the PQ is:

\[
T_{\text{query}} = \left\{ 1 + \sum_{i=1}^{\left\lceil \log \frac{C}{N}\right\rceil} D_i + D \right\} c_a + D v c_c.
\]

Let us consider two spatial data sets of cardinalities \( N_1 \) and \( N_2 \), respectively, with average node sizes \( n_1 \) and \( n_2 \), respectively. Since an SJ query between two spatial data sets is regarded to be equivalent to a set of region queries, CO for the SJ query is

\[
\text{CO} = N_1 N_2 (n_1 + n_2)^2 = N_1 N_2 \left( \frac{D_1}{N_1} + \frac{D_2}{N_2} \right)^2,
\]

where \( D_1 \) and \( D_2 \) denote the densities of the two spatial data sets. The most widely used technique for detecting an intersection between two polygons is the plane-sweep algorithm [14] whose average time complexity is \( O((v_1 + v_2) \log (v_1 + v_2)) \) where \( v_1 \) and \( v_2 \) denote the number of edges of the two polygons. Thus the exact geometry test time for the SJ query can be expressed by:

\[
\text{exact geometry test time} = (v_1 + v_2) \log (v_1 + v_2) c_c.
\]

\[\text{†} \] Due to the space limitation, we do not present an analytical formulae that estimates the overall response time of the spatial join query. However, the formula can be easily derived from [12,13].
A formula that estimates the exact geometry test time for the DR-tree is the goal to be reached. We consider that the height of the DR-tree is approximated by the parameter $g$. As described in Section 2, the value of $g$ can be obtained at around 3.

**Theorem 2.** The exact geometry test time that uses the DR-tree can be expressed by:

$$\hat{t}_{\text{exact geometry test}} = \begin{cases} (g + \frac{v_1 + v_2}{2^g})c_c & \text{for the RQ,} \\ (g + \frac{v_1}{2^g})c_c & \text{for the PQ.} \end{cases}$$  \hspace{1cm} (15)

**Proof.** The exact geometry test time is proportional to the time required to search the nodes of the DR-tree plus the sequential search time performed on decomposed components. Since the DR-tree is completely transferred into main memory when an exact geometry test is required, the time spent for the DR-tree is the CPU-time. The expected number of nodes of the DR-tree visited by the RQ algorithm is $g$, since $g$ is approximated by the length of the tree path from the root of the DR-tree. For the RQ the number of comparisons on a decomposed component is $v/2^{g+1}$, since the average number of component vertices is approximated by $v/2^g$. Similarly, for the PQ the number of comparison on a decomposed component is $v/2^g$. From Eq. (9) we obtain Eq. (15).

We now present the exact geometry test time for the SJ query involving two DR-trees. Our DR-tree join algorithm, similar to the R-tree join algorithm [15], performs a synchronous depth-first traversal of the two DR-trees. The traversal starts from the roots of the two DR-trees, then moves down the levels of the two trees in tandem until the leaf nodes are reached. Suppose main memory is large enough to load two DR-trees, the exact geometry test time of the DR-tree join can be measured by the summation of two formulas which express the respective costs for the two DR-trees as follows:

$$\hat{t}_{\text{exact geometry test}} = \left(2g + \frac{v_1 + v_2}{2^g} \log \frac{v_1 + v_2}{2^g} \right)c_c.$$  \hspace{1cm} (16)

5. **Performance evaluation**

5.1. **Evaluation of cost model**

We carried out several experiments, to compare our analytical results with experimental results using the TSIS. In particular, we focused on experiments of the refinement step. Since the evaluation of the filter step analysis was
performed by many researchers [10–13], the well-known results can be used to assess the analytical formula of the filter step. In this work we ran two sets of experiments: first, we measured the number of COs, and second, we measured the exact geometry test time.

In order to evaluate the expected number of COs, we used two groups of synthetic data sets. The first group contains 10,000 rectangles with centers following a random distribution in the work space. The second group consists of 10,000 rectangles with centers following a skewed distribution (according to the Zipf's law [16]) in the work space. The reason for using synthetic data sets is that we can control parameters such as the density and the number of rectangles. The rectangles are generated by specifying the lengths of their sides and using random number generators. The length is chosen to achieve the desired density. For the selected values of the number of rectangles and the length, $D$ ranges from 0.1 to 3.0.

The results of the first experiment are illustrated in Figs. 5 and 6. The estimated COs for each test were computed by using Eq. (8) or Eq. (13). The experimental COs for the PQ and the RQ were based on several query sets consisting of randomly generated query windows, with a side that ranged from 0% (PQ) to 4% of a side of the work space. The SJ query uses the intersection predicate, which tests whether objects in a set intersect objects in the other set. In order to obtain suitable SJ data, the original objects are taken as the first set of the join. The second set is constructed by generating 10,000 uniformly distributed rectangles. The estimation of the number of COs is very accurate, compared with the actual results using the R*-tree, since the relative error is usually below 5% and rarely above 10%.

The second experiment measures the exact geometric test time. Using Eq. (15) or Eq. (16) we estimated the exact geometry test time for the DR-tree. To
To assess the merit of the DR-tree, we also considered the exact geometry test time for the R*-tree (without the use of the DR-tree). Algorithms from the area of computational geometry, such as the point-in-polygon, rectangle-in-polygon, and plane-sweep, can be used to process the refinement step of the R*-tree. This estimate was computed by using Eq. (9) or Eq. (14).

We evaluated these analytical formulas using four real data. To get realistic results, these data were chosen from real digitized data used in existing GISs. For different object complexities, ‘Park’, ‘Lake’, and ‘State’ are considered. For a real application, ‘Map’ is extracted from a digitized map that contains soil types and land-use zones. Table 1 lists their characteristics. To describe the characteristics of the spatial objects, we provide the average number of vertices, the area of a spatial object and its MBR, and its cover characterizing the accuracy of the MBR approximation. The cover is presented by the area of the spatial object normalized to the area of the corresponding MBR. Table 1 shows impressively that the real spatial object is roughly approximated by the

<table>
<thead>
<tr>
<th>Name</th>
<th>Num. of objects</th>
<th>Distrib.</th>
<th>Num. of vertices</th>
<th>Area Object</th>
<th>Area MBR</th>
<th>Cover (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Park</td>
<td>1000</td>
<td>Uniform</td>
<td>83</td>
<td>700</td>
<td>1634</td>
<td>43</td>
</tr>
<tr>
<td>Lake</td>
<td>1000</td>
<td>Uniform</td>
<td>206</td>
<td>472</td>
<td>3105</td>
<td>15</td>
</tr>
<tr>
<td>State</td>
<td>1000</td>
<td>Uniform</td>
<td>229</td>
<td>1431</td>
<td>3456</td>
<td>41</td>
</tr>
<tr>
<td>Map</td>
<td>235</td>
<td>Real</td>
<td>88</td>
<td>429</td>
<td>1411</td>
<td>30</td>
</tr>
</tbody>
</table>
MBR. For sets of spatial objects, we additionally provide the number of objects and their distribution in the work space.

In Fig. 7, the average number of comparisons is presented for the R*-tree algorithm and the DR-tree algorithm. Due to the simplicity of the figure, the results for various region queries are not presented in this figure. We present instead the average values. The figures show that the analytical estimate is almost identical to the actual result, with a relative error usually below 10% and rarely above 15%. The figures also demonstrate that the DR-tree algorithm gives better performance than the R*-tree algorithm. The DR-tree algorithm achieves up to 86% improvement for the PQ, up to 80% improvement for the RQ, and up to 88% improvement for the SJ query. This is due to the fact that the average number of component vertices is approximated to the number of polygon vertices divided by 2^3.

5.2. Comparison with a state-of-the-art index structure

A recent work related to our TSIS is the GENESYS index structure (GSIS) [17,18]. The GSIS integrates the R*-tree and the TR*-tree. To compare the TSIS with the GSIS, we implemented both the TSIS and the GSIS. Our implementation is based on the SHORE Storage Manager (SSM). The SSM provides the R*-tree (with full concurrency control and recovery) as its basic index structure. All programs were written in C++ language on a Sun Ultra-2 workstation running on SunOS Release 5.5. For this experiment, we used four real spatial objects in Table 1.

The filter step of the TSIS and the GSIS is similar, since it is based on the R*-tree. Thus it is not necessary to compare the filter step performance of the two index structures. The refinement step of the TSIS and the GSIS, however, is performed by the DR-tree and the TR*-tree, respectively. The processing
time of the DR-tree and the TR*-tree is measured by the I/O-time spent for transferring an object into main memory plus the CPU-time spent for comparisons within the directory of a tree and for computational geometry algorithms of those components. As the performance of the refinement step strongly depends on that of the DR-tree and the TR*-tree, we explicitly measured their processing time by implementing spatial queries.

Fig. 8 shows average time required for evaluation of one query. The figure depicts the following: the horizontal axis represents the PQ, the RQ, and the SJ query. To investigate region queries with different sizes, the size of the query window varies between 10% and 50% of the data space. The vertical axis gives the time taken to perform a query.

Results show that the DR-tree is superior to the TR*-tree for the RQ and the SJ query. For the PQ, however, the performance of the DR-tree is slightly worse than that of the TR*-tree. For the interpretation of the results, the execution time differentiated by the I/O-time and the CPU-time is shown in Fig. 9. Due to its simplicity, we present the average values of region queries between 10% and 50%. This figure demonstrates that the DR-tree decreases the I/O-time, but increases the CPU-time. However, the total execution time of the DR-tree is improved for the RQ and the SJ query. The high I/O-time of the TR*-tree is caused by its large amount of storage. Due to the storage overhead, the overall performance of the TR*-tree becomes poor.

In Fig. 10, we compare storage requirements of the DR-tree with those of the TR*-tree. For different object complexities, Park, Lake, State, and Map summarized in Table 1 are considered. This figure shows that the storage cost of the TR*-tree is much higher than that of the DR-tree. The DR-tree achieves up to 71% saving in the amount of storage over the TR*-tree. In detail, the storage gain of the DR-tree in comparison to the TR*-tree depends on the
object complexity (i.e., the number of vertices). The more complex spatial objects are, the more clear the gain of the DR-tree appears. For instance, complex spatial objects such as Lake and State show a significant gain by applying our tree.

The major motivation for the development of the DR-tree is the desire to reduce the amount of memory necessary to load the tree from disk into main memory. Since the DR-tree and the TR*-tree are completely transferred into main memory when an object is required for the exact geometry test, we assumed that the main memory is large enough to read these trees. If the DR-tree

![Fig. 9. I/O-time and CPU-time.](image1)

![Fig. 10. Storage requirements.](image2)
or the TR*-tree is too large to fit entirely in main memory, then a memory allocation problem is caused by the lack of sufficient main memory. As can be seen in Fig. 10, the amount of storage of the TR*-tree can be quite large if the spatial object is complex. Copying such a large quantity of data into memory can take a long time and much memory space. It is hence essential to reduce the amount of storage of the TR*-tree, especially in a multi-user environment. The DR-tree can considerably decrease the amount of storage. In addition, the DR-tree achieves almost the same CPU performance as the TR*-tree.

Simplification of objects to be refined using the object decomposition method leads to better performance of the refinement step. However, the main drawback is caused by a number of components. Due to the generation of a large number of components, the index building time and the storage requirements become quite inefficient. However, the building time of the DR-tree takes no more than $O(dv)$ since the DMBR method needs to find DMBRs after the split, where $d$ and $v$ are the number of split-levels and of polygon vertices. Storage requirements of the DR-tree are proportional to the total number of nodes $\sum_{i=0}^{d} 2^i$ in a two-dimensional binary tree, i.e., $O(2^d)$. From experimental results, we learn that the DR-tree needs about 50% more storage than a representation of an object by point lists. Due to the appropriate index building time and amount of additional storage, the TSIS is superior to other spatial index structures.

6. Conclusions

We have proposed a TSIS based on the DMBR method. This is an extension of an existing index structure. The proposed index structure integrates two index structures, one for original objects and the other for their decomposed components. We have presented a cost model that estimates the performance of the TSIS. The main contribution of our analysis is in the refinement step. In contrast to several earlier investigations on this subject, we took into account the performance of the refinement step. Experimental results showed that the proposed cost model is accurate, the relative error being below 15% when the analytical results are compared with the experimental results. Our cost model constitutes a useful tool for spatial query optimizers that need to calculate the cost of a complex spatial query.

The performance of the TSIS was compared with the GSIS by experimental measurements. The TSIS shows almost the same performance as the GSIS, since the TSIS decreases the I/O-time but increases the CPU-time. However, the storage cost of the TSIS is much lower than that of the GSIS. The TSIS’s DR-tree achieves up to 71% saving in the amount of storage over the GSIS’s TR*-tree. Since the DR-tree and the TR*-tree are completely transferred into main memory when an object is required for the refinement step, it is essential
to reduce the amount of their storage. The DR-tree can considerably decrease the amount of storage. Therefore, the TSIS outperforms the GSIS.

The GSIS uses more accurate approximations such as the minimum bounding 5-corner and the maximum enclosed rectangle (MER). These approximations are stored in the leaf node of the TR*-tree where the corresponding MBR is already kept. The topic of this paper is the design and analysis of a spatial index structure for spatial query processing. It is not our intention to discuss which approximation is the most suitable for handling spatial objects. We only consider object decomposition techniques. As a future work, however, it is desirable to compare the DMBR approximation with more accurate approximations: 5-coner and MER.

Acknowledgements

This research was supported by the Software Technology Enhancement Program 2000 of the Ministry of Science and Technology of Korea and the Basic Research Program (grant number 99-2-315-001-3) of the Korea Science and Engineering Foundation. We wish to thank anonymous reviewers for their helpful suggestions and comments.

References