1998 International Computer Symposium
Workshop on Software Engineering and Database Systems
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

D-tree: A Bi-Level Technique for Indexing Rectangle Data in Spatial Database Systems

Kien A. Hua[1] *Ying Cai*[1] *Yu-Lung Lo*[2]

[1] School of Computer Science
University of Central Florida
Orlando, FL 32816, U.S.A
E-mail: {kienhua, cai}@cs.ucf.edu

[2] Dept. of Information Management
ChaoYang University of Technology
Taiwan, R.O.C
E-mail: yllo@mail.cyut.edu.tw

Abstract

This paper shows how a domain-decomposition hierarchy can be organized as a paginated tree with a balanced height, which can be further extended to index the data subsets (*i.e.*, first-level indexing). The data in each subset can be indexed using any existing access support structure. (*i.e.*, second-level indexing). We call this bi-level indexing structure "*D-tree*". D-tree has many desirable properties including its good data clustering characteristics, its insusceptibility to the insertion order of the data set, and its immunity to the skew in the access patterns and data distributions. We provide experimental results to demonstrate these benefits. They show that D-trees using R-trees as the second-level indices provide savings averaging 90% over using R-tree alone. In general, the proposed technique can be used to boost the performance of any existing spatial indexing methods (e.g., R*-tree, R$^+$-tree, etc.)

1 Introduction

Spatial data are expensive to retrieve due to the nature of the problem, that is, spatial data cannot be sorted or hashed like one-dimensional data. Numerous indexing techniques have been developed for spatial data in the past decades. Basically, they can be divided into two classes. The first one deals with spatial data as points in a multidimensional space. This class includes point quadtree, k-d tree, K-D-B tree, hB-tree and so forth. These structures can be classified as point trees. Normally, the shape of point trees is highly dependent on the order in which the data points are inserted. On the other hand, the second class can be called regional tree, in which spatial data are represented by intervals in several dimensions. Such spatial data normally are called regional data. For example, a rectangle is an object identified by two points in a two dimensional space. This class includes R-tree and its variants.

In this paper, we focus on rectangle data. Rectangles are often used to approximate other objects in an image for which they serve as the minimum rectilinear enclosing object. Of course, the exact boundaries of the object are also stored, but they are only accessed if greater precision is needed. The two most important types of operations on such data sets are window operations (i.e., rectangular range query) and spatial join operations. These operations are very expensive to perform. To reduce their execution time, the spatial data must be clustered. Unlike conventional textual databases, clustering in this case must base on the space occupied by the data. That is, we need to design spatial access methods that will cluster objects on disks according to their spatial locations in the space from which they are drawn. If two objects are close together in this space, they should be stored close together on disk (preferably on the same page). Many such techniques have been proposed. Among them, R-tree [1] is probably the most referenced technique in the literature. An R-tree is basically a K-dimensional variation of the B-tree [2] which indexes data consisting of intervals in K dimensions, $K \geq 1$. When $K = 2$, each node in the R-tree corresponds to the smallest rectangle that spatially encloses its child nodes. All leaf nodes appear at the same level. They contain pointers to the actual objects in the database instead of child nodes. We note that in the leaf nodes each object is represented by the smallest aligned rectangle containing it. Without loss of generality, we will assume a 2-dimensional space for the remaining of this paper.

A drawback of R-tree is that it does not result in a disjoint decomposition of space. The problem is that an object may be spatially contained in several nodes, yet it is only associated with one node. The overlap among the directory nodes, which can seriously affect its query performance, can be alleviated by "reinsertion" in R* tree [3]. However, a spatial query may still require several paths to be visited before ascertaining the presence or absence of a particular object.

R$^+$-tree [4] is another technique trying to address the overlaping problem of directory nodes. This scheme decomposes the space into disjoint bounding rectangles. Each object is associated with all the bounding rectangles that it intersects. As a result, all these bounding rectangles, represented by nodes in the R$^+$-tree, each has a path to the object. Allowing redundant paths is not performance panacea, however. It leads to an increase in the height of the tree and space overhead.

A more recent extension to the R-tree is called *Segment R-tree* (SR-tree) [5]. This scheme solves the redundant search problem associated with R-tree by linking the large objects to nodes at the higher levels of the tree. Thus, an entry in a non-leaf node can either contain a pointer to a child node, or a pointer to an actual object in the database. Objects linked to a non-leaf node are called *spanning objects* since they span at least one of the child nodes. A problem with this approach is that many entries in a node can be used up by the spanning objects. If there are many large objects in the database, the increase in the number of spanning objects will necessarily increase the height of the tree due to the reduction in the fanout of the nodes. To address this problem, it was proposed in [5] to use larger nodes at

successively higher levels. However, it was not clear how the optimal node sizes for each level could be determined. Such decisions will have to rely on the characteristics of the data set which are not usually known beforehand. Sampling was used in [5] to address this problem. It was shown in [5] that without sampling the performance of Segment R-tree is similar to that of R-tree.

Since larger objects are more likely to be retrieved by window operations, storing these objects at higher levels of the index structure, as exploited in [5], is a good idea. This property is also utilized in this paper, but without the need for sampling which is sometimes unreliable due to the skew in the distributions of the objects and/or object sizes [6]. Our technique, called *D-tree*, uses uniform node sizes, but still can maintain a maximum fanout at each tree node. In addition, we want to improve on another aspect of Segment R-tree, namely, its sensitivity to the insertion order of the data items. Depending on the insertion order, optimal data clustering cannot be guaranteed for Segment R-tree. We will show that D-tree is immune from this effect.

Unlike R-tree variants, D-tree takes a different approach to index spatial databases. In fact, it can be viewed as a data clustering technique which dynamically clusters data into disjoint subsets based on the spatial relationships among the data objects. The partitioning scheme ensures good data clustering, and allows each subset to be efficiently indexed by any existing spatial access-support structures. Without loss of generality, we use R-tree in this study to provide the second-level indexing. However, any good technique, such as R^+-tree [4], R^*-tree [3] or Hilbert R-tree [7], can also be used. The rationale for using R-tree in our study is that it facilitates some indirect comparisons with many other techniques which were also compared to R-tree (e.g., [3], [5]). In general, the proposed technique can be used to boost the performance of any existing spatial indexing methods.

The remainder of this paper proceeds as follows. In Section 2, we describe the concept of domain decomposition, and examine existing techniques for indexing the set of subdomains. The proposed scheme, *D-tree*, is presented in Section 3. Operations on D-tree are described in details in Section 4. In Section 5, we introduce a *bunch* feature which can further enhance the performance of D-tree. Our performance model and simulation results are discussed in Section 6. Finally, we give our concluding remarks in Section 7.

2 Domain Decomposition

Given a collection of rectangular objects S and their domain D, the set S can be divided into several subsets by splitting the domain D into a set of mutually disjoint subdomains. If D is split into n subdomains, D_1, D_2, ..., and D_n, then S is divided into n subsets and one spanning set accordingly. Data in each subset is fully contained in the corresponding subdomain. Data in a spanning set, however, cross at least one split line, and therefore cannot be contained in any single subdomain. We note that the domain corresponding to the spanning set is the same as the initial domain D, and is referred to as *split domain* in this paper. The domain decomposition is illustrated in Figure 1. We note that the split domain ($SplitD_i$) contains only the spanning objects (C and F) which span one or more subdomains. The set of spanning objects is called a *spanning set*. Objects enclosed in each subdomain are called *bounded objects*. The set of bounded objects associated with each subdomain is called a *bounded set*.

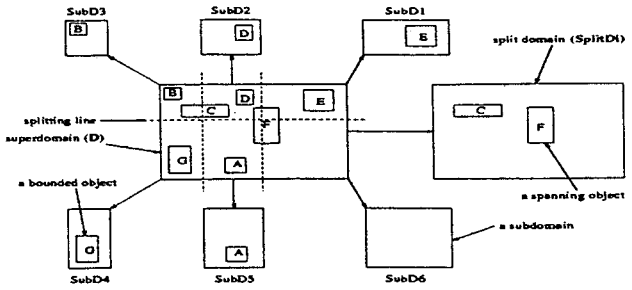

Figure 1: Domain decomposition.

Each subdomain may be recursively decomposed when its capacity exceeds a predetermined threshold, and the associated data set is divided accordingly. When a subdomain requires no further decomposition, it is at the bottom of the decomposition hierarchy and is called a *leaf domain*. As a result of the decomposition, the original data set S is partitioned into many bounded sets and spanning sets. Each bounded set is associated with a particular leaf domain while each spanning set has its own split domain. We note that any existing indexing technique can be used to index the data in each subset. D-tree, introduced in this paper, is designed to index on the subdomains. We will describe this indexing structure in more detail in the next section.

The idea of domain decomposition has been extensively used for multi-dimensional data and numerous data structures have been proposed for indexing the decomposition hierarchy. Nevertheless, these schemes are not suitable for regional data discussed here. Instead, they are designed for point data, and hence do not have to deal with spanning objects. For example, quadtree and its variations can be used for indexing the leaf domains, but they cannot deal with the split domains discussed previously. Another problem with quadtree and its variants is that, they are neigher paginated nor balanced. An advantage of a balanced index tree is the consistency in good performance for all queries independent of their access patterns.

Other index structures based on the idea of domain decomposition include k-d tree [8], K-D-B tree [9], hB-tree [10], *etc.* Again, none of these techniques are designed for regional data. Furthermore, all of these trees split a domain according to the value of the inserting data. As a result, the shape of their trees is highly dependent on the insertion order of the data points. As a contrast, we will see that different insertion orders of the data into a D-tree only result in different ordering of the entries inside the tree nodes. The shape of the D-tree is not affected by this factor at all.

3 The Structure of D-tree

For clarity's sake, we present a 2-dimensional D-tree in this paper. However, the technique can be easily extended to handle higher dimensions. A two dimensional D-tree is illustrated in Figure 2. The tree nodes can be categorized into three types according to their function:

Internal domain node: Its entries consists of a set of 2-tuples of the form (d, P), where d is a *superdomain* which spatially contains all the subdomains represented by the entries in the child node pointed at by P. For example, $d1$ is the superdomain of $d11$, $d121$, and d122, as shown in Figure 2.

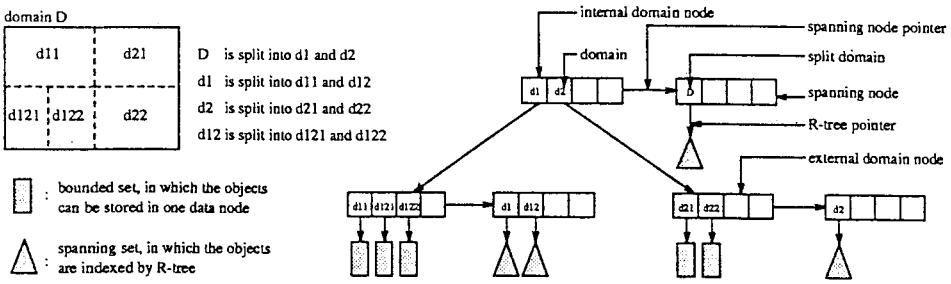

Figure 2: D-tree structure.

External domain node: Its entries are 2-tuples of the form (d, P), where d is a leaf domain and P points to a bounded set. All external domain nodes must appear at the same level, *i.e.*, the decomposition hierarchy is always balanced. The term "*domain node*" will be used when it is not necessary to differentiate between internal domain nodes and external domain nodes.

Spanning node: Associated with each domain node is a *spanning node*. Entries in a spanning node are also 2-tuples of the form (S_i, P_i), where S_i is the split domain corresponding to the ith split of the superdomain represented by the domain node having a pointer to the spanning node, and P_i is a pointer to the spanning set associated with S_i. Thus, a spanning node contains the complete history of how the corresponding superdomain was decomposed.

We note that these three types of tree nodes actually share the same data structure, which is an array of entry with the form $(Retangle, Pointer)$.

In our implementation, the objects in a spanning set are indexed by an R-tree. Each entry in a spanning node includes a pointer to the root of the corresponding R-tree. We call these trees "*spanning R-tree*" in this paper. On the other hand, since our decomposition continues until the set of objects enclosed by each leaf domain is small enough to be stored in a single page, second-level index is not necessary for a bounded set. In general, one can allow much larger bounded sets, and appropriate indexing techniques can be used to take advantage of the local characteristics of each data region.

Obviously, spatial queries can be efficiently supported by a D-tree as they involve descending the tree until the object is found in a bounded set or a spanning set. We note that the wasted search problems associated with R-tree due to the overlap of sibling nodes discussed previously are now limited to a small regional R-tree. This is one of the key properties which contribute to the good performance of D-tree. Another desirable property of D-tree is the regularity of the decomposition and the disjointness of the subdomains. This feature is particularly useful for performing set operations, *e.g.*, spatial join, as they form the basis of most complicated queries.

4 Operations on D-tree

In this section, we describe the operations designed for D-tree. They are *Search*, *Insert*, and *Delete*. The following notations are used in our discussion:

- Given an entry (D, P) in an internal domain node, $D.P$ denotes the child domain node pointed at by P.
- Given an entry (D, P) in an external domain node, $D.P$ denotes the bounded set pointed at by P.
- Given a domain node D, $D.spanning\_node$ is used to denote the spanning node corresponding to D.
- Given an entry (S, P) in a spanning node, $S.P$ is used to denote the spanning set pointed at by P.

The algorithms for the tree operations are given in the following subsections.

4.1 Search

We consider two kinds of query: *Exact Match Query* and *Range Query*. To do an exact-match search, one follows a unique path from the root of the D-tree to either a bounded set or a spanning set.

Algorithm: *Search(D_tree_root, search_object)*

1. Set *current_node* = *D_tree_root*.
2. If there is an entry, say (D, P), in *current_node* such that D contains *search_object*, then do the following:
   (a) If *current_node* is an external domain node, retrieve *search_object* from $D.P$.
   (b) If *current_node* is an internal domain node, recursively call *Search(D.P, search_object)*.
3. If none of the entries in *current_node* spatially encloses *search_object*, then do the following:
   (a) Examine the entries in the *current_node.spanning_node* from right to left until a split domain, say (S, P) is found, that spatially encloses *search_object*.
   (b) Retrieve *search_object* from $S.P$.

The search function for a range query begins the search at the root of the D-tree, and descends the tree in depth-first order to check all subdomains (including split domains) which overlap with the search window.

Algorithm: *Range_Search(D_tree_root, search_window)*

1. Set *current_node* = *D_tree_root*.
2. Examine each entry, say (S, P), in *current_node.spanning_node*. If S overlaps with *search_window*, retrieve the objects, in $S.P$, which overlap with the search window.
3. For each entry in *current_node*, say (D, P), if D overlaps with *search_window*, do the following:
   (a) If *current_node* is an external domain node, retrieve the objects, in $D.P$, which overlap with the search window.
   (b) If *current_node* is an internal domain node, recursively call *Range_Search(D.P, search_window)*.

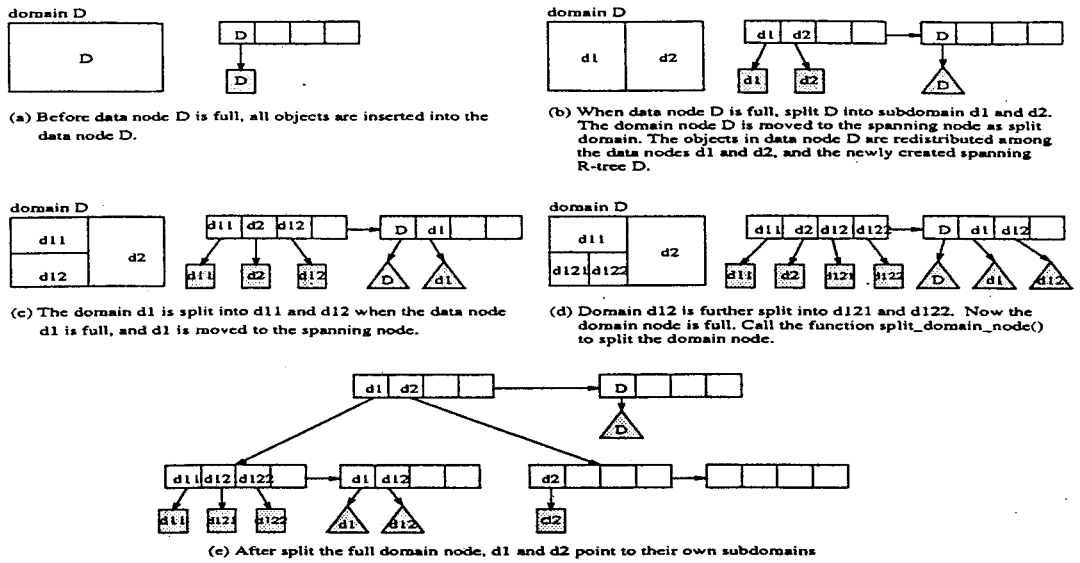

Figure 3: Split domain node.

4.2 Insert

A D-tree is initialized to have only one domain node with a single entry *(D, NULL)*, where D is the domain of the entire data set. As more data are inserted into the data set, additional domain nodes are allocated to grow the tree accordingly. The algorithm for inserting a new object is given below:

Algorithm: *Insert(D_tree_root, new_object)*

1. Set *current_node* = *D_tree_root*.
2. If there is an entry, say (D, P), in *current_node* such that D spatially contains *new_object*, then do the following:
   (a) If *current_node* is an external domain node, call *Insert_bounded_object(current_node, new_object)* to insert *new_object* into the bounded set $D.P$.
   (b) If *current_node* is an internal domain node, recursively call *Insert(D.P, new_object)* to examine the child node $D.P$.
3. If none of the entries in *current_node* spatially encloses *new_object*, then the *new_object* is a spanning object and is inserted into a spanning set as follows:
   (a) Examine the entries in the *current_node.spanning_node* from right to left until the first split domain, say (S, P) is found, that spatially encloses *new_object*.
   (b) Insert *new_object* into the spanning set $S.P$.

The insertion of an object into a bounded set can cause the corresponding data node to overflow. This condition can be handled by splitting the corresponding domain into two subdomains and creating a new split domain. The data in the old bounded set are then redistributed to the newly created bounded sets and spanning set according to their spatial properties. These operations are illustrated in Figures 3(b) and 3(c). It shows that D is split into d_1 and d_2 in Figure 3(b); and d_1 is further split into d_{11} and d_{12} in Figure 3(c). The details of this operation are described in the following algorithm:

Algorithm: *Insert_bounded_object(external_domain_node, new_obj)*

1. Look for the entry, say (D, P) in *external_domain_node*, which spatially encloses *new_obj*.
2. If P is a null pointer, create a new data node $D.P$ for D.
3. Insert *new_obj* into $D.P$.
4. If $D.P$ is full, do the following:
   (a) Split the domain D into D_l and D_r, and do the following:
      i. Create a new split domain entry (D_s, P_s) at the left-most unused entry in *external_domain_node.spanning_node* and set $D_s = D$.
      ii. Replace the entry (D, P) in *external_domain_node* by (D_l, P_l).
      iii. Create a new entry (D_r, P_r) at the left-most unused entry in *external_domain_node*.
      iv. Create two new data nodes $D_l.P_l$ and $D_r.P_r$ for D_l and D_r, respectively, and create a new spanning set $D_s.P_s$ for D_s.
   (b) For each object in $D.P$, do the following:
      i. If the object is spatially contained in D_r, then move it to $D_r.P_r$
      ii. If the object is spatially contained in D_l, then move it to $D_l.P_l$
      iii. Otherwise, insert the object into $D_s.P_s$.
5. If *external_domain_node* is full, call *Split_domain_node(external_domain_node)*

Each domain split uses one entry in the domain node. Eventually, all the available entries are used causing the domain node to overflow. Handling this condition requires the domain node to split. In the *Insert_bounded_object* algorithm, the split of a domain node is done by calling the function *Split_domain_node* in Step 5. An example is shown in Figures 3(d) and 3(e) to illustrate the split operation on a domain node. A new root is created in this example because the node being split is a root node.

Algorithm: *Split_domain_node(domain_node)*

1. If *domain_node* is the root of the D-tree, create a new root *parent* with *domain_node* as its only child; otherwise, let *parent* denote the parent node of *domain_node*.
2. Create two new domain nodes *left* and *right*, and their empty spanning nodes.

3. Look for the entry in *parent*, say (D, P), such that P points to *domain_node*. Split the domain D into two subdomains D_l and D_r, and do the following:
   (a) Replace the entry (D, P) in *parent* by (D_l, P_l), and direct P_l to point to $left$.
   (b) Create a new entry (D_r, P_r) in *parent*, and direct P_r to point to *right*.
4. For each subdomain entry (D_i, P_i) in *domain_node*, if D_i is spatially contained in D_l, it is moved to $left$; it is moved to *right*, otherwise.
5. Redistribute the entries in *domain_node.spanning_node* as follows:
   (a) Move the left-most entry in *domain_node.spanning_node* to the first unused entry of *parent.spanning_node*.
   (b) Examine the remaining entries (S_i, P_i) in *domain_node.spanning_node* from left to right, If S_i is spatially contained in D_l, it is moved to $left$.*spanning_node*; otherwise, it is moved to *right.spanning_node*.
6. Discard *domain_node* and its spanning_node.
7. If *parent* is full, recursively call *Split_domain_node(parent)*.

4.3 *Delete*

The *Delete* operation is used to remove an object from the data set. It first uses the D-tree to search for the bounded set or spanning set which spatially contains the object. This object is then removed from the set. A delete of an object from the data set can cause a subdomain to underflow. In this case, the relevant domain nodes must be merged to maintain good storage utilization. We note that a merge can also involve a split domain. For instance, in Figure 3(b), deleting the spanning objects from the spanning set corresponding to D may cause an underflow. In this case, the two bounded sets and the spanning set might have to be merged to form a single bounded set as shown in Figure 3(a). The details of the *Delete* algorithm are given in the following:

Algorithm: *Delete(D_tree_root, object)*

1. Set *current_node* = *D_tree_root*.
2. If there is an entry, say (D, P), in *current_node* which spatially contains *object*, we do the following:
   (a) If *current_node* is an external domain node, remove the object from $D.P$.
   (b) Examine the entries in *current_node.spanning_node* from right to left to find an entry, say (S, P), such that S spatially contains D. Then call *Handling_domain_underflow(current_node,S)*.
   (c) If *current_node* is an internal domain node, recursively call *Delete(D.P, object)* to examine the child node.
3. If none of the entries in *current_node* spatially encloses *object*, perform the following:
   (a) Examine the entries in *current_node.spanning_node* from right to left to find an entry, say (S, P), such that S bounds *object*. Delete *object* from the spanning set $S.P$.
   (b) Call *Handling_domain_underflow(current_node,S)*.

Algorithm: *Handling_domain_underflow(domain_node,domain)*

1. Search for the entry (D_s, P_s) in *domain_node.spanning_node* such that S = *domain*. If the spanning set is not underflow, return to the calling procedure.
2. Examine each entry in *domain_node*. Let (D_l, P_l) and (D_r, P_r) denote the two entries which were generated by splitting *domain*. Let $|D_l|$, $|D_r|$, and $|D_s|$ denote the number of objects in the data sets associated with D_l, D_r, and D_s, respectively. If $|D_l| + |D_r| + |D_s|$ objects can be stored in one disk page, do the following:
   (a) move objects from $D_r.P_r$ and $D_s.P_s$ to $D_l.P_l$.
   (b) remove (D_r, P_r) from *domain_node*, and (D_s, P_s) from *domain_node.spanning_node*.
   (c) if *domain_node* is not the root node, call *Handling_domain_node_underflow(domain_node)*.

Algorithm: *Handling_domain_node_underflow(domain_node)*

1. Let *parent* denote the parent of *domain_node*. Examine each entry in *parent.spanning_node*, starting from right to left. Let (S, P_s) denote the first entry found such that S equals the left-most split domain in *domain_node.spanning_node*.
2. Examine each entry in *parent*. Let (D_l, P_l) and (D_r, P_r) denote the two entries which were generated by splitting the domain S.
3. If $D_l.P_l$ has enough space to accommodate the entries in $D_r.P_r$, do the following:
   (a) Create a new domain node D and its spanning node *D.spanning_node*.
   (b) Move the entry (S, P_s) to *D.spanning_node* as its left-most entry.
   (c) Move all entries in $D_r.P_r$ and $D_l.P_l$ to D.
   (d) Move all entries in $D_r.P_r$.*spanning_node* and $D_l.P_l$.*spanning_node* to *D.spanning_node*.
   (e) Discard $D_l.P_l$, $D_r.P_r$, and their corresponding spanning nodes.
   (f) Remove the entries (D_r, P_r) from *parent* and (S, P_s) from *parent.spanning_node*.
   (g) Change the entry (D_l, P_l) in *parent* to (S, P), where P points at the domain node D.
   (h) If *parent* is not the root, recursively call *Handling_domain_node_underflow(parent)*; otherwise, if *parent* has only one entry, discard *parent*.

5 Performance Enchancement: Bunching Roots of R-trees

To further enhance the performance of D-tree, a *bunching* concept is used in our implementation. This strategy is illustrated in Figure 4. It shows that the roots of the three R-trees are bunched into a single page. The rationale for this feature is as follows. The root of a tree is accessed a lot more frequently than the other tree nodes. Bunching the roots of several trees, when possible, allows the access to one tree to cache several roots in the memory buffer to facilitate future accesses to the other trees. The saving in the memory space due to bunching also helps to improve the hit ratio of the memory buffer since more data can now be cached in memory. In our design, we only allow bunching be applied to tree roots associated with the same spanning node. This constraint makes it easier to determine if a bunch of trees has been examined before during the course of processing a query.

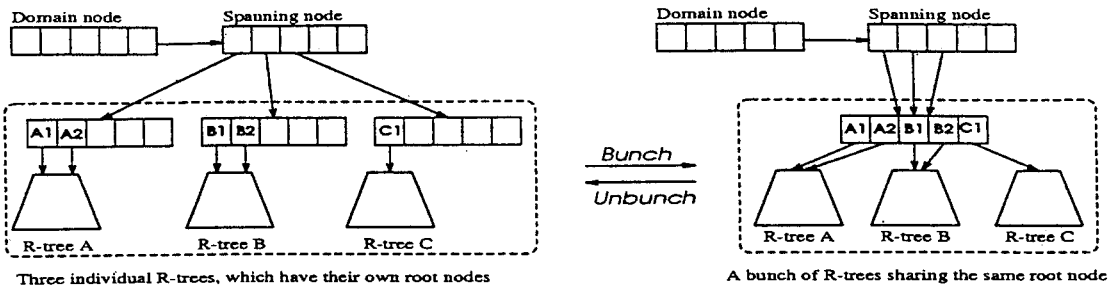

Figure 4: *Bunch_node* operation.

6 Performance Experiments

In order to assess the benefit of the proposed technique, we implemented a simulator to compare its performance with that of R-tree. The Quadratic split algorithm [1] was used to implement the R-tree since it was shown in [3] to provide better performance than Linear algorithm. We note that since the proposed technique can be used to boost the performance of any existing indexing schemes, we only considered R-tree in our performance study.

6.1 Workload Parameters

Our workload consists of a number of synthetic databases and queries. We decided to use synthetic databases since they allow us control over the different parameters that characterize the data sets. Four data sets were used in our experiments:

Data set 1: The locations of the objects in the domain space follow a uniform distribution. Their sizes also follow a uniform distribution.

Data set 2: The locations of the objects in the domain space follow a uniform distribution. Their sizes follow an exponential distribution.

Data set 3: The locations of the objects in the domain space follow an exponential distribution. Their sizes follow a uniform distribution.

Data set 4: The locations of the objects in the domain space follow an exponential distribution. Their sizes also follow an exponential distribution.

The dimensions of our domain space are (0..100K, 0..100K). There are 200K objects in each data set. These objects are inserted into a database in random order. However, the same insertion order is used for both indexing schemes (i.e., R-tree and D-tree). To generate the size for each object, we randomly generate its x-dimension and y-dimension using the appropriate distribution function. In the case of a uniform distribution, the mean size is 50; and it ranges between 0 and 100. When an exponential distribution is used, the mean size is 2,000; and it ranges between 0 and 100K. For each object, we place it at a random location in the domain space according to the appropriate distribution. Thus, in the case of a uniform distribution, the objects are spread evenly across the domain space. On the other hand, the density closer to the centroid of the domain space is denser in an exponential distribution.

In this study, we want to gain insight on the effect of the window sizes and the x-to-y aspect ratio on the indexing schemes. In our model, a query is represented by a rectangular window randomly located in the domain space. Two different groups of query sequences were considered in our study:

Group 1: Each sequence consists of 100 query windows of identical size and x-to-y aspect ratio. The query sizes are fixed for all sequences; but each sequence has a different aspect ratio. This group of queries is similar to the workload used in [5].

Group 2: Each sequence consists of 100 query windows of identical size and x-to-y aspect ratio. Both the query size and the aspect ratio are different for each sequence. This group of queries is similar to the workload used in [7].

6.2 Experimental Results

In order to facilitate the performance study, we implemented the two indexing schemes: R-tree and D-tree. Since each one of our index nodes uses one page (i.e., 1K Bytes) for storage, our performance metric is the *average I/O cost* computed as follows:

$$Avg\_IO\_Cost = \frac{\text{pages accessed by all queries in a sequence}}{\text{the number of queries in the sequence}}$$

This metric was also used in [5, 7] to study the same issue. We will also present the *Saving%*, due to the D-tree technique, which is defined as follows:

$$Saving\% = \frac{\text{Avg\_IO\_Cost(R-tree)} - \text{Avg\_IO\_Cost(D-tree)}}{\text{Avg\_IO\_Cost(D-tree)}}$$

The results of our experiments are given in the following subsections.

6.2.1 Fixed-Size Queries on Uniformly Placed Uniform-Sized Objects

In this experiment, we applied the query sequences of Group 1 to Data Set 1. The average I/O costs are shown in Table 1. We note in this table that:

$$\begin{aligned}\text{Nodes accessed by D-tree} = {} & \#\text{ domain nodes accessed} + \\ & \#\text{ data nodes accessed} + \\ & \#\text{ R-tree nodes accessed.}\end{aligned}$$

This total may not equal the sum of its parts due to the round-off errors. In our performance study, spanning nodes are counted as domain nodes to reflect the property that they are not part of the R-trees. For all query sequences, we observe that the D-tree offers tremendous improvement. It provides savings of around 90% over the R-tree. Interestingly, the data we collected for the R-tree essentially resembled those given in [5].

6.2.2 Variable-Sized Queries on Uniformly Placed Uniform-Sized Objects

In this experiment, we applied the query sequences of Group 2 to Data Set 1. The average I/O costs are shown in Table 2. We observe in this table that as we increase the sizes of the query window, the benefits of the D-tree approach decrease. This phenomenon can be explained as follows. As the size of the queries becomes large enough, a large number of objects fall into the query window causing a large number of the index nodes to be visited. Since both the R-tree and the D-tree use about the same number of nodes (6,220 nodes and 5,721 nodes, respectively, were observed in this experiment), visiting a large number of R-tree nodes or D-tree nodes results in similar I/O costs. Nevertheless, The D-tree outperforms the R-tree by a very wide margin (up to 76% saving) for more typical size queries. Since both indexing techniques use about the same number of nodes for a given database, the superiority of D-tree is attributed mainly to its better data clustering property.

6.2.3 Fixed-Size Queries on Uniformly Placed Exponential-Sized Objects

In this experiment, we applied the query sequences of Group 1 to Data Set 2. The results are given in Table 3. We observe the same behavior as seen in Table 1. However, the savings are slightly less under this workload. This is due to the fact that there are more larger objects in Data Set 2 (due to the exponential distribution) causing an increase in the number of spanning objects. As a result, the number of R-trees in the D-tree structure decreases while the size of each R-tree increases. Nevertheless, D-tree maintains savings as high as 69%. This result illustrates the capability of D-tree in adapting to the workload conditions. That is, the optimal number of R-trees used is automatically determined by the characteristics of the data set (i.e., object sizes, placement conditions, etc.)

6.2.4 Variable-Sized Queries on Uniformly Placed Exponential-Sized Objects

In this experiment, we applied the query sequences of Group 2 to Data Set 2. The results are given in Table 4. In this case, the savings due to the D-tree range between 31% and 70%. The saving is generally more when the number of nodes accessed is less. Again, this behavior reflects the excellent data clustering capability of the proposed indexing technique. Obviously, when a query accesses a very large portion of a database, the advantage of data clustering diminishes.

6.2.5 Other Experiments

More experimental results are given in Table 5, Table 6, Table 7, and Table 8. They confirm the benefits we observed for D-tree in the previous subsections (i.e., better data clustering, insusceptible to skew placement). We include these tables here for completeness. However, we will not discuss them further in the interest of brevity.

7 Concluding Remarks

We present in this paper a new indexing technique for multidimensional data called D-tree. D-tree has the following desirable properties:

- D-tree is both an indexing scheme and a dynamic data clustering technique. The adaptive data clustering mechanism ensures excellent data clustering regardless of the insertion order of the data. This feature also makes D-tree immune from the effect due to skew in the data distribution.
- Since larger objects which are accessed more frequently by window queries are stored higher in the D-tree and therefore less expensive to retrieve, D-tree performs very well even in the presence of skew in the object sizes.
- Due to the regularity of the decomposition scheme and the disjointness of the subdomains, D-tree is particularly useful for performing set operations (*e.g.*, spatial join).

A unique characteristic of D-tree is that it allows one to use different local access structures to take advantage of the regional characteristics of the data space. Our experimental results show that D-trees using R-trees as the second-level indices can provide savings averaging 90% compared to using R-tree alone.

References

[1] A. Guttman. R-tree: A dynamic index structure for spatial search. In *Proc. of the 1984 ACM SIGMOD Int'l Conf. on Management of Data*, pages 47–57, Boston, MA, June 1984.

[2] R. Baye and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An efficient and robust access method for points and rectangles. In *Proc. of the 1990 ACM SIGMOD Int'l Conf. on Management of Data*, pages 322–331, Atlantic City, New Jersey, May 1990.

[4] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A dynamic index for multi-dimensional objects. In *Proc. of the 13th Int'l Conf. on Very Large Data Bases*, pages 507–518, Brighton, England, 1987.

[5] C. Kolovson and M. Stonebraker. Segment indexes: Dynamic indexing techniques for multi-dimensional interval data. In *Proc. of 1991 ACM SIGMOD Int'l Conf. on Management of Data*, pages 138–147, Denver, Colorado, June 1991.

[6] Frank Olken and Doron Rotem. Random sampling from databases: A survey. *Statistics and Computing*, 5:25–42, 1995.

[7] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proc. of 20th Int'l Conf. on Very Large Data Bases*, pages 500–509, Santiago, Chile, September 1994.

[8] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[9] J. T. Robinson. The k-d-b tree: A search structure for large multidimensional dynamic indexes. In *Proc. of 1981 SIGMOD*, April 1981.

[10] David B. Lomet and Betty Salzberg. The hb-tree: A multiattribute indexing method with good guaranteed performance. *ACM Trans. on Database Systems*, 15(2):625–658, June 1990.

Query window	R-tree	D-tree: number of nodes accessed				
X size, Y size	Total nodes	Domain nodes	Data nodes	R-tree nodes	Total nodes	*Saving*
10, 100000	374	38	80	54	172	*117%*
31, 31622	143	16	26	25	67	*113%*
100, 10000	61	10	9	13	32	*91%*
316, 3162	37	6	4	11	21	*76%*
1000, 1000	30	6	3	8	17	*76%*
1414, 707	32	6	3	9	18	*78%*
2235, 447	32	6	3	9	18	*78%*
10000, 100	49	8	8	12	28	*75%*
31000, 31	104	12	21	17	50	*108%*
100000, 10	274	22	63	33	118	*132%*

Table 1: Performance of fixed-size queries on uniformly placed uniform-sized objects.

Query window	R-tree	D-tree: number of nodes accessed				
X size, Y size	Total nodes	Domain nodes	Data nodes	R-tree nodes	Total nodes	*Saving*
10, 10	24	6	1	8	15	*60%*
100, 10	24	6	1	8	15	*60%*
10, 100	22	6	1	8	15	*47%*
100, 100	24	6	1	8	15	*60%*
1000, 100	26	6	2	9	17	*53%*
100, 1000	28	6	2	9	17	*65%*
1000, 1000	30	6	3	8	17	*76%*
10000, 1000	58	8	13	12	33	*76%*
1000, 10000	67	8	14	17	38	*76%*
10000, 10000	144	12	64	19	95	*52%*

Table 2: Performance of variable-sized queries on uniformly placed uniform-sized objects.

Query window	R-tree	D-tree: number of nodes accessed				
X size, Y size	Total nodes	Domain nodes	Data nodes	R-tree nodes	Total nodes	*Saving*
10, 100000	502	18	33	335	386	*30%*
31, 31622	203	8	11	130	149	*36%*
100, 10000	97	6	4	57	67	*45%*
316, 3162	67	4	2	36	42	*60%*
1000, 1000	59	4	2	29	35	*69%*
1414, 707	61	4	2	31	37	*65%*
2235, 447	64	4	2	33	39	*64%*
10000, 100	94	4	4	52	60	*57%*
31000, 31	189	6	10	113	129	*47%*
100000, 10	479	10	32	294	336	*43%*

Table 3: Performance of fixed-size queries on uniformly placed exponential-sized objects.

Query window	R-tree	D-tree: number of nodes accessed				
X size, Y size	Total nodes	Domain nodes	Data nodes	R-tree nodes	Total nodes	*Saving*
10, 10	49	4	1	24	29	*69%*
100, 10	49	4	1	25	30	*63%*
10, 100	49	4	1	24	29	*69%*
100, 100	51	4	1	25	30	*70%*
1000, 100	54	4	1	27	32	*69%*
100, 1000	54	4	1	28	33	*64%*
1000, 1000	59	4	2	29	35	*69%*
10000, 1000	104	4	5	61	70	*49%*
1000, 10000	108	6	6	64	76	*42%*
10000, 10000	200	6	17	130	153	*31%*

Table 4: Performance of variable-sized queries on uniformly placed exponential-sized objects.

Query window	R-tree	D-tree: number of nodes accessed				
X size, Y size	Total nodes	Domain nodes	Data nodes	R-tree nodes	Total nodes	*Saving*
10, 100000	444	34	82	52	168	*164%*
31, 31622	175	16	28	25	69	*154%*
100, 10000	78	10	9	10	29	*169%*
316, 3162	51	6	4	9	19	*168%*
1000, 1000	40	6	3	7	16	*150%*
1414, 707	43	6	3	9	18	*139%*
2235, 447	45	6	3	9	18	*150%*
10000, 100	64	8	7	10	25	*156%*
31000, 31	146	14	20	20	54	*170%*
100000, 10	326	26	57	37	120	*172%*

Table 5: Performance of fixed-size queries on exponentially placed uniform-sized queries.

Query window	R-tree	D-tree: number of nodes accessed				
X size, Y size	Total nodes	Domain nodes	Data nodes	R-tree nodes	Total nodes	*Saving*
10, 10	35	6	1	6	13	*169%*
100, 10	35	6	1	7	14	*150%*
10, 100	31	6	1	6	13	*138%*
100, 100	34	6	1	7	14	*142%*
1000, 100	38	6	2	7	15	*153%*
100, 1000	39	6	2	8	16	*144%*
1000, 1000	40	6	3	7	16	*150%*
10000, 1000	79	8	13	12	33	*139%*
1000, 10000	85	10	14	12	36	*136%*
10000, 10000	172	14	68	18	100	*72%*

Table 6: Performance of variable-sized queries on exponentially placed uniform-sized objects.

Query window	R-tree	D-tree: number of nodes accessed				
X size, Y size	Total nodes	Domain nodes	Data nodes	R-tree nodes	Total nodes	*Saving*
10, 100000	524	16	39	345	400	*31%*
31, 31622	233	8	13	145	166	*40%*
100, 10000	108	6	5	57	68	*59%*
316, 3162	73	4	2	34	40	*83%*
1000, 1000	69	4	1	28	33	*109%*
1414, 707	73	4	2	33	39	*87%*
2235, 447	77	4	2	35	41	*88%*
10000, 100	101	4	4	51	59	*71%*
31000, 31	231	8	10	131	149	*55%*
100000, 10	474	12	28	286	326	*45%*

Table 7: Performance of fixed-size queries on exponentially placed exponential-sized objects.

Query window	R-tree	D-tree: number of nodes accessed				
X size, Y size	Total nodes	Domain nodes	Data nodes	R-tree nodes	Total nodes	*Saving*
10, 10	59	4	1	24	29	*103%*
100, 10	61	4	1	25	30	*103%*
10, 100	51	4	1	21	26	*96%*
100, 100	57	4	1	24	29	*97%*
1000, 100	62	4	1	27	32	*94%*
100, 1000	63	4	1	28	33	*91%*
1000, 1000	69	4	1	28	33	*109%*
10000, 1000	117	4	5	64	73	*60%*
1000, 10000	116	4	6	67	77	*51%*
10000, 10000	215	6	19	139	164	*31%*

Table 8: Performance of variable-sized queries on exponentially placed exponential-sized objects.