Spatial kd-Tree:
A Data Structure for Geographic Database

Beng C. Ooi
Department of Computer Science
Monash University
Clayton, Victoria
Australia 3168

Abstract

Geographic objects in two dimensional space are usually represented as points, lines, and regions. To retrieve these data objects from the database efficiently according to their spatial locations and spatial relationships, an efficient indexing mechanism is necessary. The kd-trees proposed in the literature are either unsuitable for indexing non-zero size objects such as line and region or require duplication of indexes. In this paper an alternative index structure called *spatial kd-tree* is proposed to facilitate the processing of queries concerning geographic information. The spatial kd-tree partitions a set of records on two dimensional space into small groups based on their spatial proximity. The structure not only provides efficient retrieval of objects but also maintains high storage efficiency.

Keywords:

kd-tree, data structure, associative search, geographic database.

1. Introduction

In geographic information systems(GIS), the database describes a collection of geographic objects over two particular dimensional space. Each geographic object may be classified as belonging to a particular entity class such as city, lake, road etc. These entity classes may be grouped into three generic spatial object classes namely, point, line and region. Associated with these spatial objects are alphanumeric attributes (eg. population, name, usage, etc.) that describe the aspatial characteristics of the geographic objects. Queries concerning spatial relationships among spatial objects are common in GIS, and therefore efficient query processing is very important.

In GIS, descriptive aspatial data is usually stored externally from the spatial data to allow fast retrieval of aspatial data. Using these aspatial data, the spatial relationship among spatial objects can be materialized. A typical query on spatial objects may be to "Find all cities whose population size is greater than 1000 and *the city is adjacent to a lake* whose usage is recreational". One possible way to solve the above query is to retrieve the lakes whose usage is recreational, then use each individual lake's spatial location to find all the adjacent cities whose population is more than 1000. There is no doubt that the conventional database management systems are able to retrieve the aspatial data efficiently, however, conventional database management systems can hardly be used to retrieve data based on spatial relationship. It is practically impossible to store spatial relationships among all data objects, therefore spatial relationship needs to be materialized dynamically. In order to find spatial objects by their proximity, a clustering technique for spatial objects is needed.

A number of data structures have been proposed to index multi-dimensional data [Bent75, Rob81, OuSc81, Krie82, MHN84, Gutt84]. Multidimensional B-tree[OuSc81] and k-dimensional B-tree[Krie82] are both extension of B-trees to index multidimensional data, they are not suitable for spatial objects because spatial coordinates are used as keys. Quad-trees[FiBe74], kd-trees[Bent75, Bent79], KDB-trees[Rob81] are tailored specifically for point data, which cannot be used for non-zero size objects. In [MHN84], the kd-tree is extended to cater for regions, but the indexes need to be duplicated. Grid File[NHS84] can handle point data efficiently, but it indexes non-zero size objects by transforming two dimensional spatial objects into points in higher dimensional space. R-tree[Gutt84] is a generalization of B trees for multidimensional non-zero size spatial objects, the overlaps of covering rectangles in higher level of the tree can be quite severe. In this paper, we propose a new data structure which is based on kd-tree. It is able to index regions and lines as well as points.

Structure of the original kd-tree and its variant, Matsuyama's kd-tree, are reviewed in the next section. In section 3, the new data structure for line and region objects is presented. The recursive algorithms used in searching, insertion and deletion are outlined in section 4. In the following section, implementation and future work are discussed. The conclusion is drawn in section 6.

2. Earlier kd-trees

In order to appreciate the structure proposed, it is essential to introduce the structure of the original kd-tree and the Matsuyama's kd-tree.

kd-tree[Bent75], k-dimensional homogeneous binary search tree[Knuth73], was first addressed by Bentley. A node in the tree serves two purposes: representation of actual data and direction of a search. A discriminator whose value is between *1* and *k* inclusive, is used to indicate which key the branching decision depends on. The discriminator is used cyclically, and all the nodes on the same level use the same discriminator. If more than *k* levels are necessary, then the first key will be used again in level $k + 1$. The node *P* is a data point which has two children, the left son *LOSON(P)* and the right son *HISON(P)*. If the discriminator is the *jth* attribute(key), then the *jth* attribute of any node in the left subtree is less than *jth* attribute of node *P*, and the *jth* attribute of any node in the right subtree is greater than that of node *P*. This property enables the range along each dimension to be defined, and the range is smaller in the lower level of the tree.

Since the introduction of kd-tree, many variants of kd-tree have been proposed[BeFr79, ChFu79, FBF78, Rob81]. Each of which aims to improve the performance of kd-tree on the aspects of clustering, searching, storage efficiency, balancing and query type. While most kd-trees are known to be unsuitable for indexing non-zero size objects, the kd-tree proposed by Matsuyama et al[MHN84] is tailored for non-zero size spatial objects by duplication of indexes. In the Matsuyama kd-tree, non-leaf nodes are used as directories and data is stored in the bucket indexed by the leaf node. Each bucket contains address of objects which are partially or totally included in the subspace. Region or line identifiers may be duplicated in more than one bucket. The duplication allows kd-tree to handle region data but it degrades the storage efficiency and introduces additional problems in deletion. To delete a record, it is necessary to search for all subspaces that intersect with the data object and all indexes referring to the data objects are deleted.

In the next section, we outline the structure of the spatial kd-tree which can be used to index lines and regions as well as points.

3.Spatial kd-Tree Index Structure

Irregularly shaped spatial objects can be roughly represented by various simpler techniques, two well known methods are *region decomposition* and *minimum bounding rectangle*(MBR). The region decomposition is best known with its associative structure called quad-tree [Sam84], which is a tessellation of a object into disjoint raster squares of desired resolution. The minimum bounding rectangle is the smallest box that encloses the object and it is aligned with the conventional x and y axes. While region decomposition is indispensable to image processing, MBR is useful for spatial query processing. It allows efficient processing proximity queries by preserving the spatial identification and eliminating many potential intersection tests quickly. Two objects will not

intersect if their MBRs do not intersect. This will reduce the cost since the test on the intersection of two polygons or a polygon and a sequence of line segments is expensive as compared to the test on the intersection of two rectangles. MBR can easily be defined by its *centroid* (*cx*, *cy*) and extension of each side (*dx*, *dy*) or the four coordinates (*x1*, *x2*, *y1*, *y2*).

kd-tree uses a line to partition a 2-dimensional space into two subspaces with these two resultant subspaces (*HISON* and *LOSON*) have almost the same amount of data objects. While point objects are totally included in one of the two resultant subspaces, non-zero size objects may extend over to other subspaces. Besides duplicating indexes as shown in Matsuyama's kd-tree, the alternative is to divide these non-zero size objects into subobjects which are totally included in separate subspaces. This requires complicated manipulation for storing and retrieving data objects. In our structure, spatial kd-tree, there is no division of object or duplication of index. Instead one extra line for each divided subspace is stored in the node to bound the objects' MBR whose centroid is in the subspace.

In the structure, apart from the two son pointers, discriminator and key-value(partition line), a non-leaf node has two values which specify the maximum range of *LOSON* and the minimum range of *HISON*. The maximum range value of *LOSON* is the nearest virtual line that bounds the data objects whose centroid is in the left subspace, and the minimum range value *HISON* is the nearest virtual line that bounds the data objects whose centroid is in the right subspace. Hence, internal nodes are of the form

$$(disc,\ max_{LOSON},\ loson\text{-}ptr,\ key\text{-}value,\ hison\text{-}ptr,\ min_{HISON})$$

where *disc* is a binary to indicate the dimension(0 for *x*, 1 for *y*) that is being partitioned and *key-value* is the line that partitions the space. The max_{LOSON} is the maximum range value of the *LOSON* subspace and the min_{HISON} is the minimum range value of the *HISON* subspace along the dimension specified by *disc*. The use of max_{LOSON} and min_{HISON} is realized in the next section.

Leaf nodes are of the form

$$[bound,\ min\text{-}range,\ page\text{-}pointer,\ max\text{-}range]$$

where *min-range* and *max-range* are the minimum and maximum range of objects in the page along the dimension specified by *bound*. The *bound* is always the binary complement of the parent node's discriminator *disc*. *Page-pointer* is the address of the page(bucket) in the secondary storage. In the bucket, only the object MBR and its identifier which are used to retrieve the object, are stored. Throughout this paper, object MBR and object identifier are simply referred to as a record.

Figure 1a and 1b show the structure of a spatial kd-tree and illustrate the virtual boundary(dotted line), min_{HISON} or max_{LOSON} of each resultant subspace. For easy illustration, the capacity of bucket is assumed to be 4.

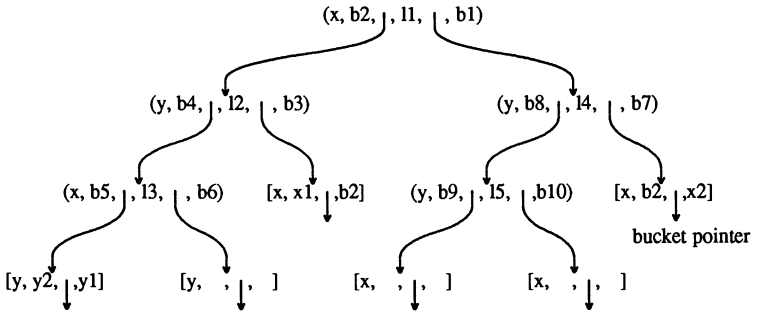

Figure 1a. The 2-d directory for spatial kd-tree.

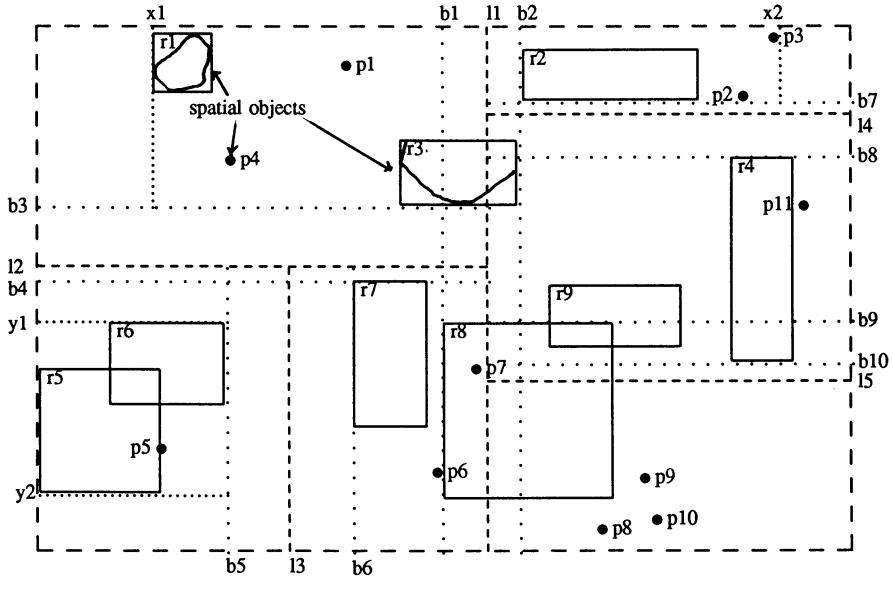

Figure 1b. The 2-d space coordinate representation.

The following definitions are required to describe the algorithms.

MAX(x, y) the function that returns the maximum value of x and y.

MIN(x, y) the function that returns the minimum value of x and y.

D[0..3]	the four coordinates x1, x2, y1, y2 (where x1 ≤ x2 and y1 ≤ y2) that defines the MBR of the data object to be inserted.
C[0..1]	the centroid ((x2 - x1)/2, (y2 - y1)/2) of MBR of the data object.
M	the maximum number of records can be stored in a bucket.
m	the minimum number of records a bucket must contain, which is usually M/2.

4. Searching and Updating

4.1 Searching

In contrast to traditional single search, we allow two types of search: *containment* search and *intersection* search. Containment search involves retrieval of data objects which are totally included in a given query region or viewing window. Intersection search comprises the containment search, which is to retrieve all data objects that intersect the query region. The only difference between these two algorithms is the area of space that the intersection testing is done. Containment search allows faster retrieval because it only searches for points and the centroid of MBR of lines and regions that are contained in the query region. Lines and regions are not contained in the query region if their centroid are not included in the query region. The existence of two search algorithms do not introduce additional complexity than that of a single search algorithm. Instead, it reduces the number of page accesses required and hence improves the retrieval process.

The containment search algorithm is outlined as follows:

```
C_SEARCH(node, map_space)
/* the initial inputs are the root of the tree and the
   four coordinates that describe the entire map */
if node = leaf_node then
      map_space[node.bound*2] = node.min-range;
      map_space[node.bound*2 + 1] = node.max-range;
      if INTERSECT(map_space) then
            CHECK_LEAF(node);
      return;
/* rsubspace[0..3] and lsubspace[0..3] consist of x1, x2,
   y1 and y2, to describe two resultant subspaces */
for i = 0 to 3 do
      rsubspace[i] = map_space[i];
      lsubspace[i] = map_space[i];
lsubspace[node.disc*2 + 1] = MIN(node.key_value, node.max_LOSON);          (1)
```

```
 /* to update upper bound of LOSON */
rsubspace[node.disc*2] = MAX(node.key_value, node.min_HISON);                (2)
 /* to update lower bound of HISON */
if INTERSECT(lsubspace) then
      C_SEARCH(node.loson-ptr, lsubspace);
if INTERSECT(rsubspace) then
      C_SEARCH(node.hison-ptr, rsubspace);
```

Presumably, the query region which has the same structure as D is defined globally, and the boolean function *INTERSECT* returns *TRUE* if the query region intersects with its argument and *FALSE* otherwise. The function *CHECK_LEAF* is to fetch the page addressed by the leaf node, and retrieve the records that are in the query region.

In containment search, the smaller of the values, maximum range (max_{LOSON}) and *key-value* is used to determine whether the record may be in the *LOSON*, and the larger value of minimum range (min_{HISON}) and *key-value* is used for the *HISON*. In intersection search, the search space is larger than the search space in containment search. To perform this, the MIN function equation 1 is simply replaced by assignment of $node.max_{LOSON}$, and the MAX in the equation 2 is replaced by the assignment of $node.min_{HISON}$.

4.2 Insertion

Inserting index records for new data objects is similar to insertion in point kd-tree where, as new index records are added to the bucket, the bucket is split if it overflows. At each node, the algorithm determines the branching direction and updates the boundary if MBR of the object extends over the boundary. The process of searching is done recursively and on reaching the leaf node, the bucket is fetched and insertion may be performed.

```
INSERT(node)
/* the data structure D and C are declared globally
   and the initial input is the root of the tree */
if node = leaf_node then
      if there is room in the bucket then
            insert the record;
      else
            call SPLIT(node);
      return;
if C[node.disc] ≤ node.key_value then
      if D[node.disc*2 + 1] ≥ node.max_LOSON then
```

```
            node.max_LOSON = D[node.disc*2 + 1 ];
        INSERT(node.loson-ptr);
    else
        if D[node.disc*2 ] ≤ node.min_HISON then
            node.min_HISON = D[node.disc*2 ];
        INSERT(node.hison-ptr)
```

In order to add a new record to a full bucket containing M records, it is necessary to divide these $M + 1$ records into two buckets. The division is performed on the dimension which is the larger of the subspace's rectangle. Objects are ordered according to ascending centroid coordinate along the dimension to be partitioned. The objects are divided into two groups, and the line that partitions the rectangle may be anywhere between these two groups. One of the resultant buckets will contain $m + 1$ records, and the other will contain $M - m$ records. The strategy is outlined as follows:

```
SPLIT(node)
Get the dimension of the longer side of the subspace;
Order objects in the ascending order of their centroid along the dimension to be partitioned;
Partition the objects halfway between object m+1 and m+2;
Create a new leaf and new non-leaf to index the resultant two buckets;
```

4.3 Deletion

The deletion of a record may cause the page to underflow, if the number of records is less than half of the page capacity, and this may degrade the storage efficiency. To ensure a better storage efficiency the underflowed page is merged with the page indexed by its neighboring leaf node, and the resultant page is resplit if overflow occurs. However, this is not possible if the neighboring node is not a leaf node. In this case, the following strategy would be employed. The algorithm involves deletion of the leaf node and the parent non-leaf node, the pointer which points to the parent node is redirected to the neighboring node. As the consequence, the subspace in the neighboring node is expanded, and the records in the deleted leaf node need to be reinserted into the neighboring node. In the example shown in figure 2, assuming that the bucket indexed by the leaf node 3 is underflowed, then node C is deleted and is replaced by node D. Once the records in the deleted bucket are reinserted into the subtree of node D, leaf node 3 and the bucket can be disposed. The method not only ensures a sufficiently high storage efficiency, but it also attempts to maintain the balancing of the tree. A rather different reinsertion strategy is employed in R-tree[Gutt84], whereby subtrees indexed by the entries of the deleted node are reinserted from the root. Since R-tree is a

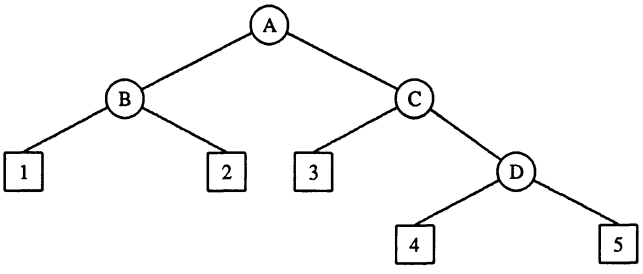

Figure 2. Example on Deletion.

height-balanced tree, the entries from higher-level must be placed higher in the tree.

4.4 Static Tree Construction

Usually, the tree is initially built from an available database, then the tree would be constructed statically instead of dynamically. Dynamic construction refers to the process of inserting all the records in the database into an initially empty kd-tree by means of the insertion algorithm. Static construction is a pre-processing technique which packs the tree such that all buckets are as full as possible. Two ways of building the tree statically are top-down and bottom-up construction. Top-down construction of kd tree starts from the root, recursively, partitions the leaf-node till it has no more than M records. Bottom-up construction partitions the records to form all leaf nodes, then form the parent node for each pair of leaf nodes, and recursively working upwards till the root is formed. Top-down strategy may not guarantee a very good storage efficiency while bottom-up strategy may require a completely different set of codes to implement the algorithm. However, the top-down strategy will guarantee the storage efficiency to be comparable with that obtained by bottom-up construction, if the partition is done in such a way that the two resultant subspaces have the amount of records which is in the multiple of some integral number of pages.

To accommodate static construction, the SPLIT needs some modification to split $M + 1$ or more records. The records are stored in a linked list indexed by the leaf node, they are flushed into the bucket only when the list contain less than M records.

5. Discussions

Some of the fundamental spatial operations[SDMD87] performed in GIS are adjacency, proximity(within), containment, intersection, and distance(nearest and furthest). Although these operations are semantically different, they can generally be transformed into a sequence of containment searches and intersection searches. Detail of implementation of each operation is not within the scope of this paper.

If GIS is the only supported system, then it is important that the directory is stored in the main memory to avoid page access on the directory. Unfortunately, the directory may have to be stored in the secondary storage if the load of the system is high. This can be done as suggested in [Knuth73](section 6.2.4). A family of internal nodes which is a subtree, is stored in the same page in secondary storage. The page contains the root is usually stored in the main memory. The following example illustrate the secondary storage structure of spatial kd-tree, where each secondary page can hold 8 nodes.

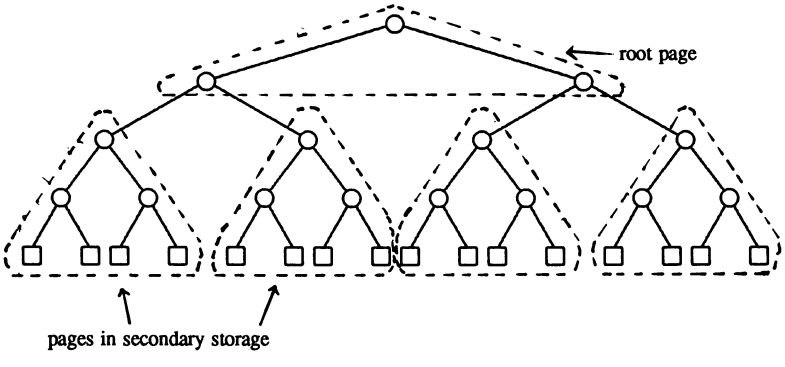

Figure 3. Storage Structure.

At the writing of this paper, the simulation of performance on spatial kd-tree and R-tree is being conducted. Two important measurements such as storage efficiency (defined by the ratio of number of buckets used and the number of bucket is actually needed) and query efficiency (defined by the average number of bucket accessed) will be used by the simulation to compare the effectiveness of each structure.

6. Conclusions

Previously, the kd-tree has been considered as an unsuitable indexing mechanism for non-zero size spatial objects. However, a new data structure which was proposed shows that it is possible to extend the kd-tree definition to overcome this problem.

In summary, the proposed spatial kd tree avoids the problems of index duplication[MHN84] and object division. Two different algorithms are proposed for searching. Although two algorithms are required, they do not incur higher complexity than a single search strategy. Furthermore, they prune the search space more efficiently. In addition, a better deletion strategy which ensures good storage efficiency is proposed. Although the structure is proposed to index geographic data, it can easily be extended to index general multidimensional spatial objects.

Acknowledgments. The research was sponsored by Monash Graduate Scholarship. I am grateful to Guat Khunn Goh and Hock Thiam Ch'ng for carefully reading the initial versions of this paper and making valuable suggestion for improving the presentation of this paper. The comments from Dr K.J. McDonell, Dr B. Srinivasan and the referees are gratefully acknowledged.

References

[Bent75] Bentley, J.L.
Multidimensional Binary Search Trees Used for Associative Searching, Comm. of ACM, Vol. 18, No. 9, 1975, pp 509-517.

[Bent79] Bentley, J.L.
Multidimensional Binary Search Trees in Database Application, IEEE Trans. on Soft. Eng., Vol. SE-5, No. 4, 1979(July), pp 333-340.

[BeFr79] Bentley, J.L. and Friedman, J.H.
Data Structures for Range Searching, Computing Survey, Vol. 11, No. 4,(Dec) 1979.

[ChFu79] Chang, J.M. and Fu, K.S.
Extended K-D Tree Database Organization: A Dynamic Multi-Attribute, Clustering Method", IEEE Compsac, 1979, pp 39-43.

[FiBe74] Finkel, R.A. and Bentley, J.L.
Quad Trees: A Data Structure for Retrieval on Composite Keys, Acta Informatica 4, 1974, pp 1-9.

[FBF78] Friedman, J.H., Bentley, J.L., Finkel, R.A.
An Algorithm for Finding Best Matches in Logarithmic Expected Time, Trans. on Math., Vol. 3, No. 3,pp 1978, pp 209 - 226

[Gutt84] Guttman, A.
R-Trees: A Dynamic Index Structure for Spatial Searching, SIGMOD 84, pp 47-57.

[Knuth73] Knuth, D.E.

Sorting and Searching in the series of The Art of Computer Programming, Vol 3, Addison Wesley, Reading 1973.

[Krie82] Kriegel, H.P.
Variants of Multidimensional B-Trees as Dynamic Index Structure for Associative Retrieval in Database System, Proc. of the 8th Conf. on Graphtheoretic Concepts in Computer Science, Hanser Publishing Company, pp 110 - 128, 1982.

[MHN84] Matsuyama, T., Hao, L.V. and Nagao, M.
A File Organization for Geographic Information Systems Based on Spatial Proximity, Computer Vision, Graphic, and Image Processing 26, 1984, pp 303-318.

[NHS84] Nievergelt, J., Hinterberger, H. and Sevcik, K.C.
The Grid File: An Adaptable, Symmetric Multikey File Structure, ACM Trans. on Database Systems, Vol. 9, No. 1, 1981, pp 38 - 71.

[OuSc81] Ouksel, M. and Scheuermann, P.
Multidimensional B-Trees: Analysis of Dynamic Behavior, BIT 1981, pp 401 - 418.

[Rob81] Robinson, J.T.
The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes, SIGMOD, 1981, pp 10 -18.

[SDMD87] Sacks-Davis, R. and McDonell, K.J.
GEOQL - A Query Language for Geographic Information Systems, Forthcoming Monash Technical Report, 1987.

[Sam84] Samet, H.
The Quadtree and Related Hierarchical Data Structures, ACM Computing Survey, Vol. 16, No. 2, June 1984, pp 187-260.