Bitmap R-trees

C.H. Ang, T.C. Tan
School of Computing
National University of Singapore
Republic of Singapore, 119260
E-mail: {angch, tantc}@comp.nus.edu.sg

Abstract

Bitmap R-tree is a variant of R-tree in which bitmaps are used for the description of the internal and the external regions of the objects in addition to the use of minimum bounding rectangles. The proposed scheme increases the chance of trivial acceptance and rejection of data objects, and reduces unnecessary disk accesses in query processing. It has been shown that with the bitmaps as filters, the reference to the object data file can be cut down by as much as 60%.

1 Introduction

With the widespread use of computers, many non-conventional applications have been developed to handle spatial data in two and three dimensions. The spatial data include map objects such as bridges in a Geographical Information System (GIS), or electronic components such as transistors in a VLSI Computer Aided Design (CAD) application.

With the ever increasing demand for the manipulation of spatial data, spatial database system evolves and many data structures have been proposed such as quadtree [5], kd-tree [4], and R-tree [7]. (Refer to [10] for more details on the various spatial data structures that have been in use.) As one of the earliest proposed tree structures for nonzero-sized spatial objects, R-tree has always been used as a yardstick in assessing the performance of other related data structures. Variants of the R-tree, such as R^+-tree [11], R^*-tree [3], and Rr-tree [9] have been proposed all with the aim to improve performance.

All these structures can be used to support various forms of query, in particular, point query and region query. The bitmap R-tree we propose in this paper aims to enhance the query processing performance of R-tree by extending it with two types of bitmaps. It is hoped that with the use of these bitmaps, some unnecessary disk accesses can be avoided. Even though the construction of the bitmaps requires some processing time, the overall saving in query processing can still be gained.

The paper is organized as follows. We begin with a discussion on R-tree and its variants in section 2, followed by introducing bitmap R-tree in section 3. Query processing using bitmap R-tree is described in section 4 and its performance analysis based on empirical results is presented in section 5. In section 6 we draw our conclusion.

2 R-tree and its variants

R-tree is a multi-dimensional generalization of B-tree [2]. It is used as an indexing structure to speed up the retrieval of spatial objects. It is height-balanced and the insertion and deletion of an object may trigger node splitting and merging.

Usually, a k-dimensional spatial object is fully described with all its spatial and aspatial attributes of interest contained in a long record in a data file. In order to access this record quickly, its offset from the beginning of the file is used as an index which is stored in a leaf node of an R-tree. An entry in a leaf node of an R-tree is a tuple (mbr, oid), where mbr is the k-dimensional minimum bounding rectangle of the object, and oid is the object identifier that can be used to retrieve the full object description record from the file.

Each entry in a non-leaf node of an R-tree is a tuple $(mbr, childptr)$, where $childptr$ is a pointer to a lower level node in the R-tree, and mbr is the minimum bounding rectangle (MBR) that covers all the rectangles in the lower level node pointed to by $childptr$. Figure 1 shows the MBR of an object. Figures 2 and 3 show the planar view of an R-tree, and its corresponding structure, respectively.

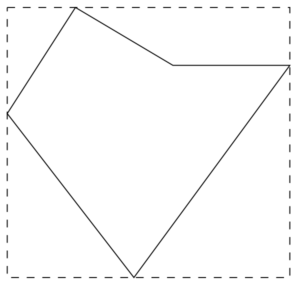

Figure 1: MBR of an object.

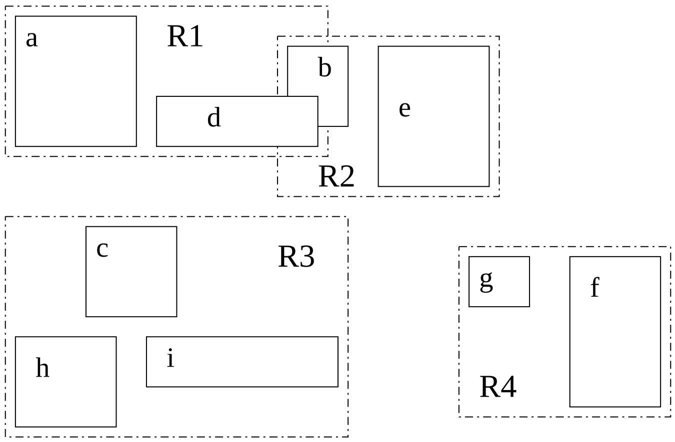

Figure 2: Planar view of an R-tree.

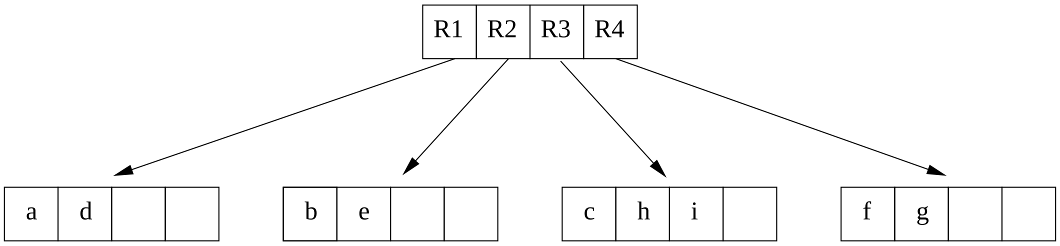

Figure 3: Structure of an R-tree.

Each R-tree node has a predetermined capacity which is the maximum number of entries a node can carry. During the insertion of an entry, a node will be split when it overflows. A new node is allocated, and all entries in the overflowed node together with the new entry are redistributed into the overflowed node and the new node according to some rules. A new index which contains the MBR of the new node and the pointer to the new node has to be inserted into the parent node (the node that contains the pointer to the overflowed node). The overflowed node's MBR has to be updated. The insertion of the new index may in turn cause the parent node to split and the node splitting may propagate upwards. If the propagation reaches the root node, the root node is split, a new root node is created and the depth of the R-tree is increased.

Overlapping regions exist not only in the leaf node but also in the non-leaf nodes of an R-tree. When a query point falls within an overlapping region, there will be multiple access paths leading to the desired objects, resulting in higher search cost [8]. To overcome this problem, R^+-tree was proposed. It uses the object clipping approach to divide objects into as many sub-objects as required so that the MBRs of these internal nodes do not overlap each other. The disjoint partitioning of subspaces ensures a single search path for a given query point.

In Rr-tree, in addition to the use of MBR, the maximum internal rectangle (MIR) is used and stored in the leaf node together with the MBR of an object. The MIR is the largest rectangle that is strictly contained in the object. Any point falling inside the MIR of an object is definitely within the object. The use of MIR can therefore further reduce the number of accesses to the data file, especially in answering a point query. Figure 4 shows the MBR and MIR of an object.

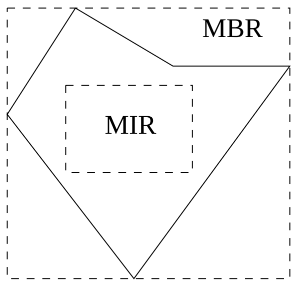

Figure 4: MBR and MIR of an object.

3 Bitmap R-tree

The basic idea of using bitmaps in bitmap R-tree is similar to the Rr-tree except that instead of storing the MIR, the internal and external bitmaps of the object are stored together with its MBR at the leaf node.

A spatial object is usually represented by a simple polygon (a polygon without holes). Two regions are defined by the polygonal boundary of the object: one that is strictly contained in the object and another that is strictly outside the object but still within the MBR. Instead of using 16 bytes for the description of MIR as is done in the Rr-tree, we could make good use of these spaces to describe the two regions in the highest resolution attainable, resulting in the use of two bitmaps, the *internal bitmap* and the *external bitmap*.

Each MBR is divided into 8 by 8 grid cells. A small bitmap can be constructed and stored in 8 bytes if a cell is represented by a bit. The 1s in an internal bitmap of a spatial object represent the corresponding grid cells that are completely within the object while the 1s in an external bitmap represent the corresponding grid cells that are completely outside the object but inside the region confined by the MBR. Figures 5, 6, and 7 show a triangle that is superimposed on an 8x8 grid, its internal bitmap, and its external bitmap. From the internal bitmap and the external bitmap, it is not difficult to obtain the outline bitmap which shows all grid cells that intersect with the boundary of the object. The three bitmaps are closely related; given any two, we can easily derive the remaining bitmap.

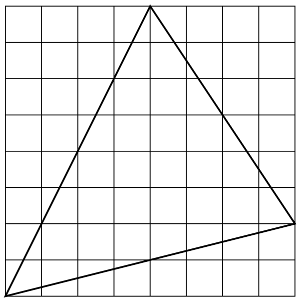

Figure 5: A triangular object T.

1	1	1	0	0	1	1	1
1	1	0	0	0	0	1	1
1	1	0	0	0	0	0	1
1	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1

Figure 6: External bitmap of T.

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0
0	0	0	1	1	0	0	0
1	0	0	1	1	1	0	0
0	0	1	1	1	1	1	0
0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 7: Internal bitmap of T.

The algorithm to generate the external bitmap of a polygon is adapted from the polygon filling algorithm [6] used in graphics rendering. The outline bitmap is generated by using the voxel traversal algorithm [1] used in ray tracing. With these two bitmaps, the internal bitmap can be obtained.

The deletion and insertion of an object into a bitmap R-tree is the same as that for an R-tree except that the corresponding bitmaps are to be created during the insertion of an entry at the leaf node level.

4 Query processing

In an R-tree, if the query is an arbitrary region, the MBR of the query region will be used to select object entries that satisfy the query MBR. In addition, the object descriptions of all the selected entries are needed for further testing with the query region.

With a bitmap R-tree, the internal and the external bitmaps of the query region, denoted *Qintmap* and *Qextmap*, can be generated and used in query processing. Using the bitmaps and the MBR of the query region, certain entries may be trivially accepted or rejected without further testing.

Table 1 lists the tests on the bitmaps that are used in processing a query.

In point query, we are to locate all objects that contain a given point. In an R-tree, even though the given point is contained in an MBR, we cannot determine if the object really contains the point. In a bitmap R-tree, if the point is found to be in the *intmap*, then it is inside the object and the object is trivially accepted. If the point is in the *extmap*, then it is outside the object and the object is trivially rejected. Of course, if it is not within the MBR, then it is also outside the object. Only when the given point is found to be in the MBR of an object and is neither in its *intmap* nor its *extmap*, then will the object description record stored in the data file be accessed.

In a region query, we want to find all objects covered completely by the given region (a subset query), containing it (a superset query), or simply intersecting it (an intersection query). We only consider intersection query in our study.

In an R-tree, the region specified is usually a rectangle. Any MBR in a leaf node found intersecting with the specified region will cause the corresponding object record to be read in. In a bitmap R-tree, if any portion of the *intmap* of an object is found to be within

Test	**Result**
PointExt	If a query point is mapped into a bit in *extmap* with value 1, the point is in the external region and the object is trivially rejected.
PointInt	If a query point is mapped into a bit in *intmap* with value 1, the point is in the internal region and the object is trivially accepted.
BmpNonIntersection	If none of the 0s in *extmap* is mapped to any of the 0s in *Qextmap*, the regions do not intersect. Otherwise the regions may intersect.
BmpContainment	If all the 0s in the *extmap* are mapped to 1s in *Qintmap*, it implies that the spatial object of the entry is strictly contained in the query region.
BmpIntersection	If any of the 1s in *intmap* is mapped onto a 1 in *Qintmap*, the two regions intersect. The object can be trivially accepted for the intersection query.
LineIntersection	Clip the query line to the MBR of the selected entry. Create bitmap of the clipped line and perform BmpIntersection test.

Table 1: Bitmap test functions

the region, the object is trivially accepted. If the region is found to intersect only with the *extmap* of an object, then the object is trivially rejected. Otherwise, the object record has to be retrieved for further checking.

The use of bitmaps of the given query region simplifies the filtering step. The cost of checking the intersection between the bitmaps of the given region and the bitmaps of the objects is low. Therefore specifying a non-rectangular query region in a bitmap R-tree will not pose any problem in the performance of query processing.

5 Performance analysis

We implemented R-tree and bitmap R-tree. We did not implement Rr-tree as the area of an MIR is usually smaller than that of the corresponding *intmap*, and with the use of the *extmap*, the number of objects that can be trivially accepted or rejected based on the use of both the *intmap* and the *extmap* should be more than that with the use of MIR.

The space requirements of R-tree and bitmap R-tree can be worked out easily. The sizes of an MBR, a child pointer, and a coordinate are 16, 4, and 4 bytes respectively. Thus the sizes of an entry in a leaf node in an R-tree and bitmap R-tree are 20 and 36 bytes respectively. The sizes of the entry in an internal node of both structures are 20 bytes. Assuming that a node is of 1K bytes, then the capacity of a leaf node in an R-tree and bitmap R-tree are 50 and 28. The capacity of a non-leaf node in both trees are 50.

The size of the domain from which the spatial objects are drawn is 100000 by 100000. The spatial objects are no larger than 1000 by 1000. The objects are made up of randomly generated points, lines, triangles and quadrangles. Equal number of each type of objects were generated.

During the experiment, the total number of objects trivially accepted and rejected, and the total number of disk accesses were noted. The number of input objects varied from 10000 with increment of 10000 till 30000. The node size varied from 1K bytes to 4K bytes, with increment of 1K bytes at each step. Three different sets of 100 query objects of the same type (point, line, etc.) were randomly generated. The 100 points were used in point query. The line objects were used to locate objects that intersect with the query lines. The quadrangles were also used for intersection query. This would allow us to find out the effect

Node size	Point query	Line int.	Quadrangle int.
10000	68	69	69
20000	69	65	63
30000	69	64	62

Table 2: Efficiency (in percentage) of bitmaps

Node size	Point query	Line int.	Quadrangle int.
1K	36	27	21
2K	30	20	13
3K	24	15	9
4K	20	5	7

Table 3: Percentages of saving in disk accesses using bitmap R-tree

of the shapes of the query objects on the efficiency of the bitmaps in the query processing.

The *efficiency* of the bitmap is defined as t/c, the ratio between t, the number of candidates that are trivially rejected or accepted, and c, the number of candidates selected by checking their MBRs with the query point or with the MBR of the query region. For the same set of test data and the same query set, the efficiency of the bitmaps remained very much stable regardless of the size of the nodes. When the same query set was used, the efficiency varied slightly when different sets of test data were used. This can be seen from Table 2. In point query, 68% of those candidates were either trivially rejected or accepted. For intersection query, the efficiency is 62%.

Although the efficiency of the bitmaps seemed to be quite high, the reduction in disk accesses was about 36% at best. The percentages of saving in disk accesses are shown in Table 3.

The saving is large for point query using small nodes but the advantage of using bitmap R-tree wanes when the node size is increased. This is because a bigger node allows more objects to be stored, hence the MBRs of the internal nodes are larger, more regions overlap, causing more nodes in lower level to be read in the search. As the capacity of the leaf nodes of the bitmap R-tree is smaller, the same set of candidates are stored in more leaf nodes.

Therefore more leaf nodes have to be accessed even though the efficiency of the filter remains the same.

We have experimented to find out the efficiency of the *intmap* and the *extmap* of the internal nodes. The results showed that they were not effective at all (less than 0.2%) in trimming the access path. This is indeed not surprising. In fact, *intmap* is useless as the low level node is still required to be read in when the query point is found within the *intmap*. The only saving can only come from the use of *extmap* to terminate the search from proceeding further. But when the MBR of an internal node is represented by an 8 by 8 bitmap, a grid cell is very likely covered by some of the rectangles of the corresponding entries stored in the low level node. This means that the external bitmap will contain mostly 0s.

6 Conclusion

The use of MBR in *R*-tree helps to filter away objects that are unlikely to be included in the answer to a query. Although the use of MBR is simple, it may be too crude at times, especially when its external region is large.

By using MIR in *Rr*-tree, even though it helps to filter additional candidates that are definitely to be included in the answer set, it is still not effective when the external region of an object is large and thus the corresponding MIR is small. Since each MIR requires 16 bytes, the capacity of a leaf node in an *Rr*-tree is significantly reduced.

We propose to enrich the *R*-tree structure with additional information that can quickly differentiate if a given point is inside or outside an object. With the same storage overhead as MIR by using 16 bytes for storing the internal bitmap and the external bitmap of a given object in bitmap *R*-tree, we are able to trivially accept or reject more than 60% of the candidates identified based on the MBRs alone. An object with a large external region has a large external bitmap, and hence this increases its chance of being rejected in point query or region query.

Although a bitmap *R*-tree needs more space and time to process the bitmaps and the tree resulted may be deeper due to the larger number of leaf nodes each of which having fewer entries, the empirical results show that the query performance has generally improved by more than 20% in terms of the number of disk accesses. Since the CPU processing time is only a small fraction of disk access time, a net gain in query performance is evident.

References

[1] John Amanatides and Woo, A fast voxel traversal algorithm for ray tracing, EUROGRAPHICS (1987).

[2] R. Bayer, E. McCreight, Organization and maintenance of large ordered indices, Acta Informatica 1, 3(1972), 173-189.

[3] N. Beckmann, H. Kriegel, R. Schneider, B. Seeger, The R^*-tree: An efficient and robust access method for points and rectangles, Proc. of the ACM SIGMOD Conference, Atlantic City (1990), 322-331.

[4] J. L. Bentley, Multidimensional binary search trees used for associative searching, CACM 18, 9(1975), 509-517.

[5] R. A. Finkel, J. L. Bentley, Quad Trees, a data structure for retrieval on composite keys, Acta Informatica, 4(1974), 1-9.

[6] J.D. Foley, A. Dam, S.K. Feiner, and H. F. Hughes, Computer Graphics: Principles and Practice, 2nd edition.

[7] A. Guttman, R-tree: A dynamic index structure for spatial searching, Proc. of the ACM SIGMOD Conference, Boston, (1984), 47-57.

[8] E. G. Hoel, H. Samet, A qualitative comparison study of data structures for large line segment databases, Proc. Int. Conf. on Management of Data (1992).

[9] J. Kim and H. Bae, The design of efficient access method for objects, GIS: Technology and applications, Proc. for the Far East Workshop on Geographic Information Systems, Singapore, 21-22 June 1993, 91-105.

[10] H. Samet, The Design and Analysis of Spatial Data Structures, Addison-Wesley, (1989).

[11] T. Sellis, N. Roussopoulos, C. Faloutsos, The R^+-tree: A dynamic index for multi-dimensional objects, Proc. 13th International Conference on Very Large Data Bases, Brighton, England, (1987), 507-518.