QR-tree: a hybrid spatial data structure *

Y. Manolopoulos[1] E. Nardelli[2] A. Papadopoulos[1] G. Proietti[2]

[1] Dep. of Informatics, Aristotle University, 540 06 Thessaloniki, Greece
e-mail: {manolopo,apapadop}@athena.auth.gr

[2] IASI, National Research Council, 00185 Roma, Italy and
Dep. of Pure & Applied Mathematics, University of L'Aquila, 67100 L'Aquila, Italy
e-mail: {nardelli,guido}@iasi.rm.cnr.it

Abstract

The need for efficient and effective manipulation of spatial data is obvious in applications that handle very large volumes of spatial objects. In this paper, we propose a hybrid data structure to index and manipulate point and non-point objects. This hybrid structure is a combination of two famous and well-studied data structures, i.e. the quadtree and the R-tree. The basic idea is to partition the dataspace by means of the quadtree and to associate an R-tree with each partition. We give insertion, deletion and range search algorithms as well as we present experimental results based on computer generated data and real data from the Sequoia 2000 and TIGER projects. We evaluate the performance of the new structure in comparison to that of the original R-tree structure and we discuss structure improvements, advantages and disadvantages.

Keywords: spatial data indexing, quadtree, R-tree, secondary storage, performance

1 Introduction

The need for efficient and effective indexing and manipulation of spatial objects arise in many applications such as: Computer-Aided Design (CAD), Computer Vision and Image Processing, Robotics, Cartography and GIS [Laur92]. However, although the above applications have many similarities, each one has its own specialties and irregularities, that must be taken into account in order to design and implement a Spatial Database Management System (SDBMS). Spatial data management is an active area of research and a large number of very interesting and promising results have been proposed in the literature over the past ten years. Here, we focus on the physical level of such a SDBMS, which deals with the storage and indexing of the underlying dataspace. More specifically, we are interested in the secondary storage organization, since it is clear that the vast volumes of data under consideration need to reside in disk.

We can classify the spatial objects into two main categories. In the first class we have objects with zero extend (i.e. points) and in the second, objects with non-zero extend (i.e. line segments, polygons, rectangles). Some of the proposed methods that manipulate points are the Bucket PR quadtree [Same90], the Grid File [Niev84] and the hB-tree [Lome90]. For an excellent survey we prompt to [Same90]. However, the need for indexing arbitrary objects lead to the design of data structures capable of handling efficiently non-point objects as well as point objects. Among these structures we highlight the MX-CIF quadtree [Abel83, Same90], the Cell-tree [Gunt89], the

*Research performed under the 4th Italian-Greek bilateral scientific protocol

LSD-tree [Henr89], and the R-tree [Gutt84]. By R-tree we denote an important family of access methods which include the packed R-trees [Rous85], the R^+-tree [Sell87], the R^*-tree [Beck90], and the Hilbert R-tree [Kame94]. The members of the R-tree family use rectilinear minimum bounding rectangles (MBRs) to approximate the objects. The search is performed in two steps. First, a set of candidates is retrieved using the approximation of the objects. Second, the set of candidates is tested using the actual geometry of the objects, in order to determine which of the objects are indeed part of the answer. This paradigm is called filter-refinement [Oren86], and is a common approach in order to avoid complicated and time consumable computations during index lookups.

Our hybrid access method comprises of two parts, a main memory high-level structure and a disk resident low-level structure. The new organization also follows the filter-refinement paradigm, since we make use of R-trees for the low-level structure. The main-memory structure is a quadtree which consists of a few levels only and we associate a whole R-tree to each quadtree leaf node. Thus, the quadtree is mainly used to partition the dataspace and to assign R-trees to portions of it. Although the partitioning is regular, the structure performs better than the R-tree, and therefore it is competitive with more sophisticated members of the R-tree family. However, the structure has some other interesting advantages that we discuss later in this paper.

The rest of the paper is organized as follows. In Section 2 we give some background on the quadtree and R-tree data structures, and the motivation behind the proposed approach. In Section 3 we present the proposed method and the algorithms for insertion, deletion and range searching. Section 4 contains the experimental results and performance comparisons. Finally in Section 5 we conclude the paper.

2 Background and Motivation

The quadtree is a hierarchical data structure which is widely used to store images. Each node has exactly four children corresponding to the NorthWest, NorthEast, SouthWest and SouthEast quadrants. A variation of the structure called Bucket PR-quadtree [Same90] was designed to enable the manipulation of point objects in secondary storage. Each portion of the dataspace corresponds to a bucket and when the population of a bucket exceeds the bucket capacity, we split the bucket in four. The situation becomes more complicated if the objects have non-zero extend. In this case an object may intersect several bucket boundaries. One quadtree variation introduced to handle rectangular objects is the MX-CIF quadtree [Abel83, Same90].

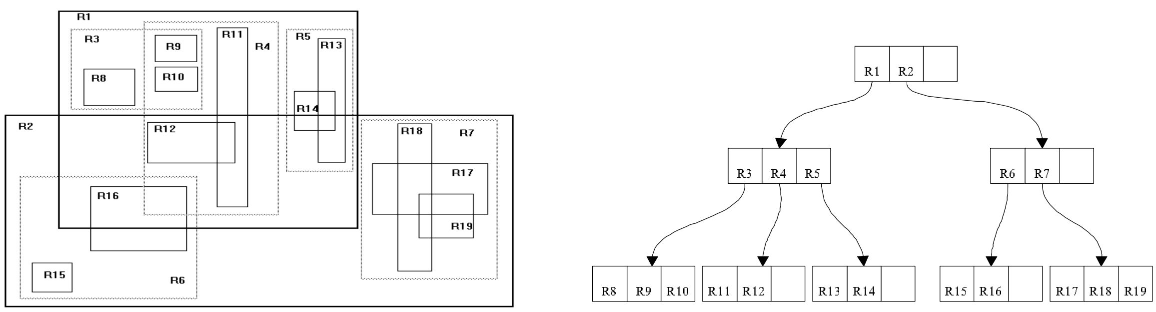

Figure 1: A sample of rectangles in the dataspace and the corresponding R-tree.

The R-tree [Gutt84] is a hierarchical data structure, designed for use in secondary storage, and it is a generalization of the B-tree for multidimensional spaces. An example is presented in Figure 1. The structure handles objects by means of their conservative approximation. The most simple approximation of an object's shape is the Minimum Bounding Rectangle (MBR). Each node of the tree corresponds to exactly one disk page. Internal nodes contain entries of the form (R, *child-ptr*), where R is the MBR that encloses all the MBRs of its descendants and *child-ptr* is the pointer to the specific child node. Leaf nodes contain entries of the form (R, *object-ptr*) where R is the MBR of the object and *object-ptr* is the pointer to the object's detailed description. Since MBRs of internal nodes are allowed to overlap, we may have to follow multiple paths from root to leaves when answering a range query. This inefficiency triggered the design of the R^+-tree [Sell87] which does not permit overlapping MBRs of internal nodes.

One of the most important factors that affects the overall structure performance is the node split strategy. In [Gutt84] three split policies are reported, namely exponential, quadratic and linear split policies. However, more sophisticated policies that reduce the overlap of MBRs are reported in [Beck90] and in [Kame94] that lead to R^*-trees and Hilbert R-trees respectively. These two structures are the most robust and efficient, to the best of the authors' knowledge. In this paper, we base our work on the original R-tree structure [Gutt84] for simplicity. Any other spatial data structure can be used as the low level structure.

Although the R-tree has been proven to be a very promising and efficient structure, there are some hot spots that may cause deterioration of the overall performance:

- Since MBRs may overlap, additional disk accesses may be required, because multiple paths may have to be examined in order to answer range and exist queries.
- As the population of the stored objects increases, overlapping increases too.
- We may be forced to examine several levels of the tree although there are no (or a few) objects satisfying the query.
- As the number of nodes increases (and possibly the tree grows taller), insertions of new objects require more disk accesses and more CPU time, in order to specify the most promising tree path to store an object.
- The performance of the structure can vary substantially under different insertion orders of the objects. In static environments, a careful preprocessing and specific ordering of the data objects can offer a high degree of performance improvement [Rous85,Kame93]. On the contrary, in full dynamic environments such a preprocessing seems impossible, because we have no complete knowledge of the objects involved.
- Operations such as the spatial join and the map overlay are very costly to perform [Brin93]. Moreover, we may be forced to retrieve a single page several times, in order to test all candidates.

The issues mentioned above affect the performance of the R-tree structure for point and non-point objects. In the next section we introduce our hybrid data structure based on a combination of quadtrees and R-trees, we try to soften the inefficiencies mentioned above and we examine the advantages and disadvantages of such an approach.

3 The Proposed Approach

3.1 Structure Presentation

The main idea behind the proposed method, is to perform a partitioning of the dataspace first and index the objects with respect to their position in space. However, an arbitrary distribution of the objects is not expected to perform well, because in order to answer a query we have to examine all the trees. What we need, is a careful object-to-index assignment according to space occupancy, in order to visit as few tree indices as possible. The proposed structure is a combination of the quadtree and the R-tree. We name this combination a QR-tree. The quadtree is used as the high-level structure to partition the dataspace into several regions. Then we use a forest of R-trees, as the low-level structures, to index the objects that are distributed over these regions of the dataspace.

Definition: We say that a quadtree is of height H, when the root appears at level 0 and the leaves at level H.

To get a better understanding of the structure we give an example in the following figures. In Figure 2, the dataspace contains points, whereas in Figure 3 it contains rectangles. For simplicity reasons, the height of the quadtree is 1 and its nodes are depicted by circles. The triangles under the quadtree nodes correspond to the R-trees, which we use to store the objects. Thus, each R-tree is associated to exactly one quadtree leaf node.

According to Figure 2, objects 1 and 2 are stored in the R-tree associated to the NorthWest quadrant, whereas objects 3 and 4 are stored in the R-tree associated to the SouthWest quadrant. As we can easily observe, objects 6 and 7 intersect the boundaries of several regions. In this case we clip these objects and store the pieces into the corresponding R-trees. Observe, also, that a point object is always stored, without clipping, in an R-tree which corresponds to the leaf level of the quadtree. This is because we can uniquely associate a point to a quadrant all the way from the root down to the leaf level. For example, we may assume that quadrants are closed with respect to bottom and left boundaries and open with respect to top and right boundaries. For the rest of this section we assume that the quadtree is complete, for illustration purposes. Structure tuning and generalizations are presented in Section 5.

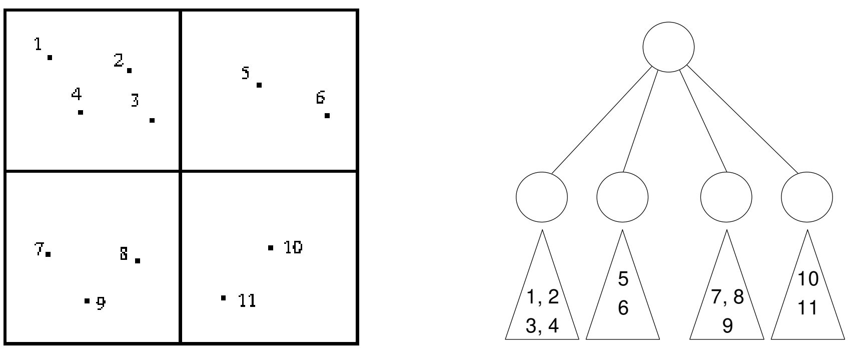

Figure 2: Associating point objects to R-trees.

The distribution of objects with respect to their spatial location has some additional interesting consequences:

- objects that are far from each other, are stored (in most cases) into different tree indices thus, reducing the overlap.

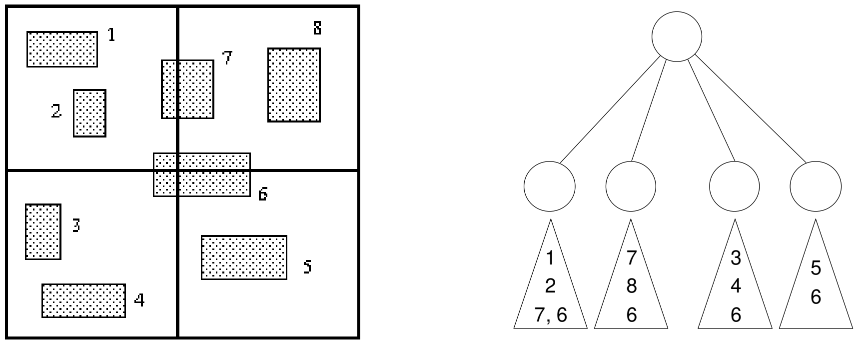

Figure 3: Associating non-point objects to R-trees.

- the affect of the insertion order in performance is reduced, because a specific object is associated with only one index (irrelevant to the insertion order).
- access to least populated dataspace regions is faster, reducing the average response time of range and exist queries.
- the spatial join operation can be performed more efficiently because we can combine the corresponding quadrants.

The basic question that arises, is how many quadtree levels we need. Assume we have a dataspace $D_x \times D_y$ in the 2-dimensional Euclidean space. The objects of the underlying dataspace are rectilinear rectangles. In most applications the maximum area of an object rarely exceeds the 1% of the dataspace area [Page93]. Let us assume that we have a complete quadtree of height 4. Thus, the total number of quadtree nodes is $(4^5 - 1)/(4 - 1) = 341$, whereas the total number of quadtree leaf nodes is $4^4 = 256$. Clearly, the root corresponds to the entire space $D_x \times D_y$. Each child of the root covers a $D_x/2 \times D_y/2$ area. Each leaf node covers a $D_x/16 \times D_y/16$ area, which corresponds to 0.4% of the dataspace. Therefore, it is reasonable to claim that:

- further partitioning may lead to high redundancy, especially when the area of the objects is near the maximum value.
- the storage overhead of a 4-level quadtree is negligible and therefore we can keep it in main memory.

The first observation above is justified by the following two tables, which contain statistics on a number of maps depicting countries of Africa and countries of Europe. Initially these maps were in vector-based format and transformed into 512×512 raster-based images. Afterwards, maps were implemented as region quadtrees. Each table contains three columns. The first column presents the quadtree leaf node dimensions, the second the percentage of leaf nodes and the third the number of nodes with the corresponding dimensionality. As we can observe, the majority of the information is concentrated near the pixel level. Therefore, the finer the granularity the larger the number of leaf (external) nodes and therefore, the more the redundancy introduced by the clipped objects.

The storage overhead of the whole quadtree is negligible for reasonable granularities. We need to store only the leaf nodes of the quadtree. The structure can be organized as an array

Quadtree node dimensions	Percentage (%) of leaf nodes	Number of nodes
1x1	59.2927	4839.7358
2x2	23.5380	1921.2735
4x4	10.5114	857.9905
8x8	4.1283	336.9716
16x16	1.6810	137.2075
32x32	0.6304	51.4528
64x64	0.1741	14.1075
128x128	0.0418	3.4150
256x256	0.0023	0.1886
512x512	0.0000	0.0000

Table 1: Maps of countries of Africa.

Quadtree node dimensions	Percentage (%) of leaf nodes	Number of nodes
1x1	51.6611	1937.3070
2x2	25.5647	958.6816
4x4	12.4535	467.0112
8x8	5.9724	223.9661
16x16	2.7262	102.2338
32x32	1.1275	42.2816
64x64	0.3847	14.4253
128x128	0.0890	3.3380
256x256	0.0209	0.7830
512x512	0.0000	0.0000

Table 2: Maps of countries of Europe.

of quadrants. For each array position we maintain two values: a) the locational code [Abel84] of the corresponding quadtree block and b) the pointer to the underlying R-tree structure. For a quadtree of height 4 we need 8 bits to identify all the leaf quadrants (256 in number). So, we assign one byte for the locational code and four bytes for the pointer to the R-tree. Thus, we need $256 \times 5 = 1280$ bytes in total.

Let us now present the algorithms for insertion, deletion and range searching. The details of the algorithms concerning a specific R-tree can be found in [Gutt84].

Algorithm Insert - INS
Input: the root of the quadtree (R) and the new object (E).
Output: nothing.
[INS1] Determine which quadrants are intersected by E.
[INS2] Decompose E into a set P of pieces according to the quadrants it intersects.
[INS3] For each $p \in P$, insert p into the R-tree associated to the quadrant.

Algorithm Delete - DEL
Input: the root of the quadtree (R) and the object to be deleted (E).
Output: if success return TRUE, else FALSE.
[DEL1] Determine which quadrants are intersected by E.
[DEL2] Decompose E into a set P of pieces according to the quadrants it intersects.
[DEL3] For each $p \in P$, delete p from the R-tree associated to the quadrant.

Algorithm Range Search - RS
Input: the query window (QW) and the root of the quadtree (R).
Output: the objects that overlap QW.
[RS1] Determine which quadrants are intersected by QW.
[RS2] Decompose QW into a set Q of pieces according to the quadrants it intersects.
[RS3] For each $q \in Q$, perform a range search into the R-tree associated to the corresponding quadrant.
[RS4] Collect all answers and remove duplicates.

3.2 Analysis of Redundancy

In this simple analysis we estimate the average number of pieces per object and the total number of objects imposed by clipping. We assume that objects are distributed uniformly with respect to the centroid of their MBRs and that the quadtree portion of the QR-tree is complete with leaves appear at level H. Let o_x and o_y denote the average x and y extends of the objects. First we derive a formula that estimates the probability that an object will not be clipped. In Figure 4 we have a quadtree of height 1 (for simplicity), and therefore has four leaf quadrants. All quadrants are equal in size and let r denote the side of each. From the construction process of the QR-tree we have that $o_x \leq r$ and $o_y \leq r$. In case an inserted object is not clipped, its centroid must fall inside the shaded area of Figure 4(a).

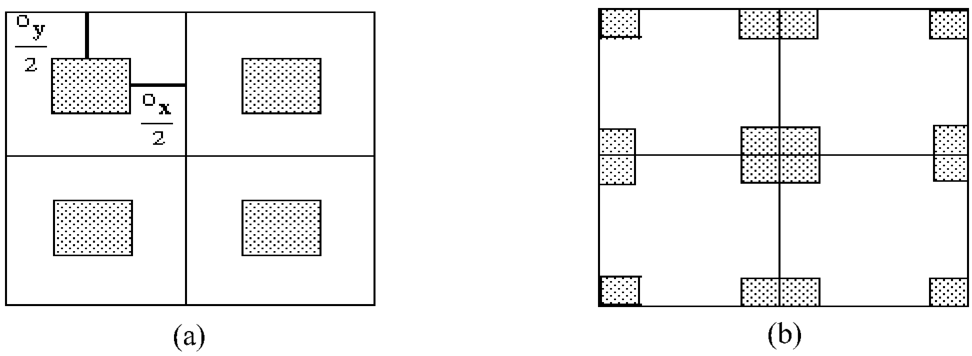

Figure 4: If the center of an object falls inside the shaded area, the object is not clipped (a) and is clipped in four (b).

Therefore, the probability for a non clipped object is:

$$Prob[NC] = Shaded\_Area/Dataspace\_Area = \#quadrants * (r - o_y) * (r - o_x)/Dataspace\_Area$$

If the dataspace is normalized to the unit square we get:

$$Prob[NC] = \#quadrants * (r - o_y) * (r - o_x) \tag{1}$$

Observing Figure 4(b) we get that if we allow an object to wrap around the dataspace boundaries, the probability for an object to be clipped in four is equal to the shaded area. Therefore:

$$Prob[C4] = \#quadrants * o_x * o_y \tag{2}$$

Thus, the probability for an object to be clipped in two is:

$$Prob[C2] = 1 - Prob[NC] - Prob[C4] = 1 - \#quadrants * [(r - o_y) * (r - o_x) + o_x * o_y] \tag{3}$$

Using Equations (1-3) we can estimate the average number of pieces per object. We have:

$$\begin{aligned} AVG_{pieces} &= 1 * Prob[NC] + 2 * Prob[C2] + 4 * Prob[C4] \quad \Rightarrow \\ AVG_{pieces} &= 2 * \#quadrants * o_x * o_y - \#quadrants * (r - o_y) * (r - o_x) + 2 \end{aligned}$$

If H denote the level where the quadtree leaf nodes reside, then

$$\#quadrants = 4H \qquad\qquad r = 1/2H$$

Therefore the above equation becomes:

$$AVG_{pieces} = 2 * 4H * o_x * o_y - 4H * (1/2H - o_y) * (1/2H - o_x) + 2$$

and after some algebraic manipulations we get:

$$AVG_{pieces} = 4H * o_x * o_y + 2H * (o_x + o_y) + 1 \tag{4}$$

If N denotes the number of inserted objects, then clearly, the total average number of objects is:

$$AVG_{Total\_Objects} = AVG_{pieces} * N \tag{5}$$

Thus, if the height of the quadtree and the average x and y extends of the objects composing the dataspace are known, then one could estimate the average number of pieces per object, and therefore the overhead introduced by redundancy by using the simple formulae (4) and (5).

4 Experimental Results

We implemented the QR-tree and the R-tree (with quadratic split) data structures in C programming language under UNIX, and run the experiments on DEC 3000 and SUN SPARC workstations. Assuming that all numbers and pointers require 4 bytes, each entry of a node requires 16 bytes to store the MBR, 4 bytes to store the pointer to the next level and 4 bytes for the object ID. For each node we further need 4 bytes for the pointer to the father, 1 byte to store the father entry, 1 byte for the type of the node (i.e. root, leaf, or internal) and 1 byte to store the number of occupied entries. The maximum capacity of a n R-tree node was set to 66 and the minimum to 33. The quadtree portion of the new structure and the root of the R-tree were kept in main memory. The height of the quadtree was set to 4.

The first set of experiments involved point objects and the second non-point objects. For point objects we used 100,000 computer generated points with uniform distribution. Also we used 62,562 real data points (California places) from the Sequoia 2000 benchmark [Ston93] ftped from site s2k-ftp.cs.berkeley.edu. For non-point objects we used 100,000 computer generated rectangles having their centroid obey a uniform distribution, whereas their size was taken to obey a uniform distribution between 0 and 1% of the dataspace area. Also, we used real datasets

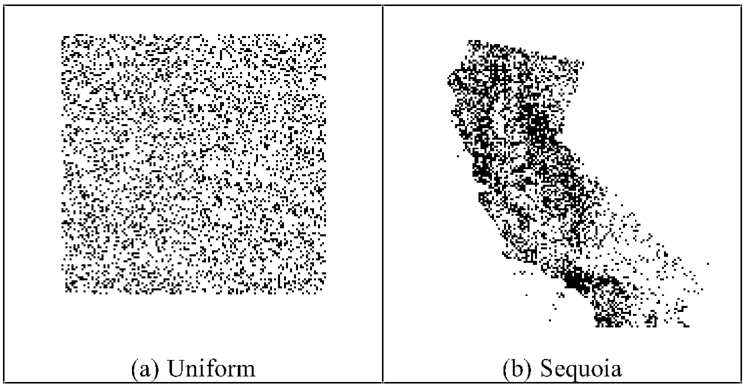

Figure 5: Samples of point data.

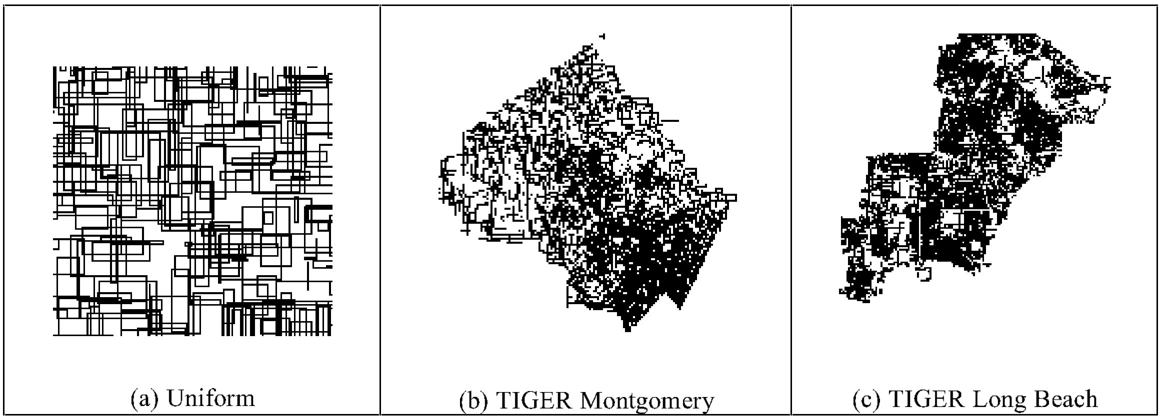

Figure 6: Samples of rectangle data.

from the TIGER project [TIGE94]. More specifically, we used the Montgomery county (39,323 objects) and Long Beach (53,146 objects) datasets. Samples of the different datasets used are presented in Figures 5 and 6.

The dataspace dimensions were set to $1,000 \times 1,000$. The window query area was set to vary from 5×5 to 200×200. Each window query was performed 1,000 times and the average number of page accesses was measured. An assumption that we followed was that a specific disk page is retrieved only once for each operation, since we may have an adequate buffer space to store the R-tree full path from root to leaves.

We also measured the performance of insertions. For each experiment we inserted into the structures the 90% of the total number of objects and we measured the number of node splits occurred, during the insertion of the last 10% of the total number of objects.

In the following we present the results of the various experiments. The graphs in Figure 7 include the information concerning the number of disk page accesses during range searching. Each graph presents the relative performance of QR-tree with respect to R-tree. Let $DA(R)$ and $DA(QR)$ denote the average number of disk accesses for R-tree and QR-tree respectively. Then, the relative performance of QR equals $\frac{DA(QR)}{DA(R)} \times 100$. In Table 3 we give the performance measures for point enclosure queries (stabbing queries), and finally in Table 4 we present comparison data for the case of insertions. Each cell of this table contains the total number of disk accesses plus the number of node splits during insertions for the two structures (first value for the R-tree, second value for the QR-tree).

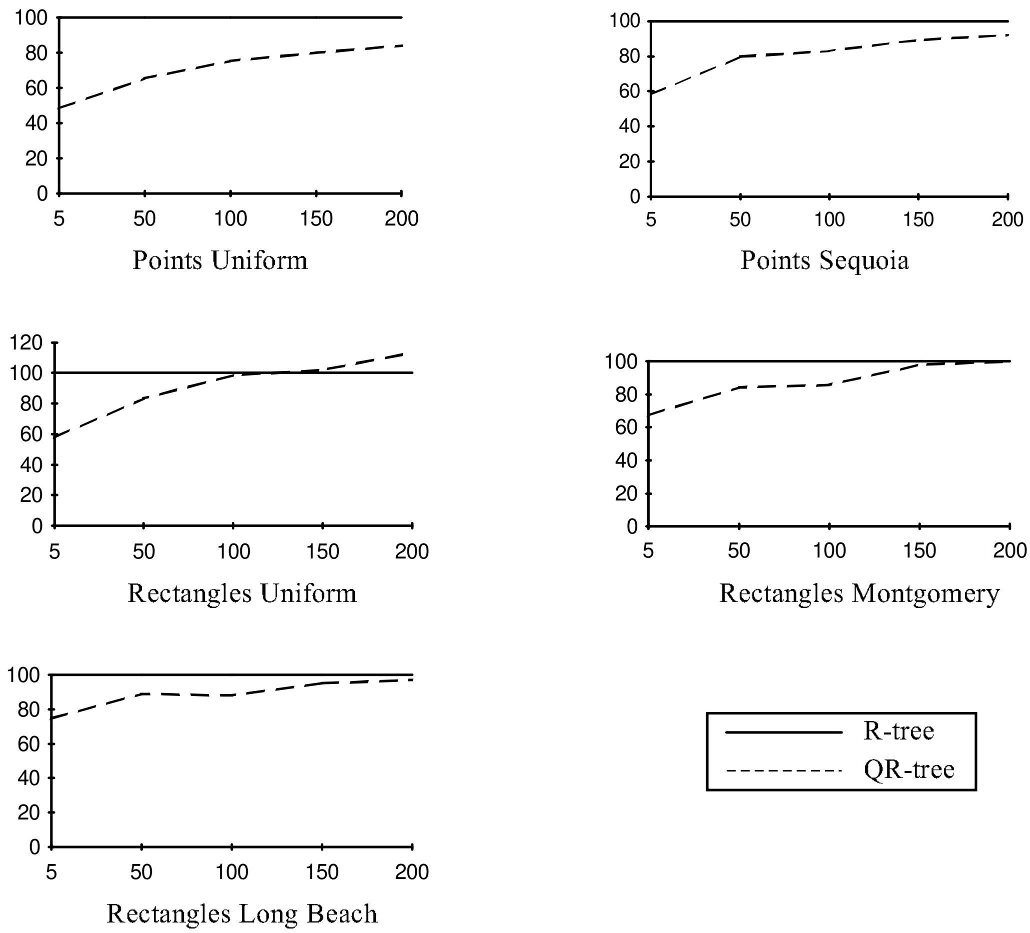

Figure 7: Relative range query performance. Disk accesses Vs. query window side.

Studying the experimental results of Figure 7 and Tables 3 and 4 we observe the following:

- The performance gain in disk accesses for window queries ranges from 5% (large queries) to 50% (small queries). The gain is grater for point datasets. For small query windows the gain is more significant and therefore the QR-tree is suitable for this type of queries. For large query windows the performance of the QR-tree is near the performance of the R-tree and in some cases (Rectangles Uniform) R-tree wins marginally.
- The performance of point enclosure queries (Table 3) shows that the gain ranges from 19% (Rectangles MG) to 40% (Rectangles Uniform).
- During insertions (Table 4) the number of splits occurred in the structure is generally

	Rects. Uniform	Rects. MG	Rects. LB
QR-tree	9.27	1.79	1.63
R-tree	15.44	2.20	2.10

Table 3: Point enclosure query performance. Average number of disk accesses.

	Points Uniform	Points Sequoia	Rects. Uniform	Rects. MG	Rects. LB
QR-tree	78	129	428	84	108
R-tree	214	165	250	107	113

Table 4: Number of node splits during insertion of the last 10% of objects.

greater for the R-tree. Therefore, the QR-tree is suitable in dynamic environments where insertions and deletions are frequent.

As we observe, the need for more powerful partitioning is evident, in order to be more competitive. Some considerations are discussed in the next Section.

5 Discussion and Conclusions

In the present report we have proposed a new hybrid structure which is a combination of the popular quadtrees and R-trees, where the quadtree part serves as an index toward a collection of R-trees. This structure may be used for the storage and manipulation of point and region data. The space requirements of the quadtree portion are negligible, e.g. one disk page. Thus, in essence, a single R-tree is replaced by a great number of R-trees under a divide-and-conquer philosophy. Insertion, deletion and range searching algorithms are given. Experimental results are included, which show an improvement over the R-tree performance, in general.

We have implemented R-trees as the low-level structure. Equally well we could have implemented any other member of the R-tree family, such as the R^+-tree or the R^*-tree, and combine these structures with the quadtree structure too. We anticipate that in such a case, similar performance improvements could be achieved by the hybrid structure produced.

5.1 Structure Improvements

We have chosen the quadtree as the high-level structure, due to its simplicity and because we wanted to study the behavior of regular decomposition against various distributions of data objects. The results show that there is an improvement, but in order to increase the gain in disk accesses several improvements must be consider. So far we assumed that the quadtree portion of the structure is complete (i.e. all leaves appear at the same level). This is expected to perform well for uniform or near uniform object distributions. However, when the distribution is skew some regions may contain two few objects or may be empty and the performance of the structure may deteriorate. We can follow several directions in order to compensate :

a) more sophisticated splitting mechanism

Following this direction, we do not split in advance. Initially the quadtree portion contains only one node, the root of the quadtree which corresponds to the entire dataspace. As objects are inserted into the structure, we split the region in four only when the number of objects inserted into the quadrant is grater than the bucket capacity. Since we restrict the quadtree to fit in only one page, this policy is continued until the growth of the quadtree reaches the bucket capacity. After this threshold, we do not split further and store the incoming objects into the R-trees associated to the regions. This policy enables the construction of a non complete quadtree, following the object distribution up to a point.

b) structure reorganization

If insertions and deletions are not frequent, a structure reorganization is feasible. Empty regions are discarded and more populated regions are subdivided further. Also, since the objects are distributed to several R-tree structures, a local reorganization could be applied without "freezing" the whole structure. Moreover, packing [Rous85, Kame93] can be applied to individual R-trees.

c) other combinations

Finally, the performance of other combinations of structures must be evaluated. For example, instead of QR-tree we could design a KD/R-tree or any other valid combination.

5.2 Spatial Joins

Assume we have two spatial relations R and S. A complex operation occurring often in spatial applications is the spatial join with respect to a join predicate. Here we discuss the intersection spatial join: *given two spatial relations* R *and* S*, find all pairs* (x, y) *with* $x \in R$ *and* $y \in S$ *such that* $x \cap y \neq \emptyset$. Using QR-trees to index the spatial relations, we have a prior knowledge about the intersecting candidates. We give an example in Figure 8.

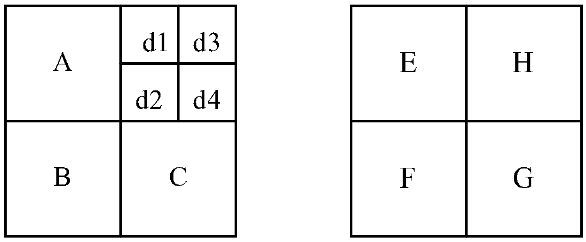

Figure 8: Relations organized as QR-trees.

Observing Figure 8, we see that in order to perform the intersection spatial join between relations R and S, we have to combine region A with E, B with F, C with G and finally d_1, d_2, d_3, d_4 with H. Due to object clipping, no other combination is needed. This property of QR-trees can help in cost reduction during spatial joins.

Further work include the implementation of the spatial join operation in order to study the efficiency of QR-trees against other structures such as other R-tree variants or Z-order [Oren86] and also the design of algorithms for nearest neighbor, topological and other types of queries.

References

[Abel83] D.J. Abel and J.L. Smith: A data structure and algorithm based on a linear key for a rectangle retrieval problem, *Computer Vision, Graphics and Image Processing*, Vol.24, No.1, pp. 1-13, 1983.

[Abel84] D.J. Abel: A B$^+$-tree structure for large quadtrees, *Computer Vision, Graphics and Image Processing*, Vol.27, pp. 19-31, 1984.

[Beck90] N. Beckmann, H.P. Kriegel and B. Seeger: The R*-tree: an efficient and robust method for points and rectangles, *Proceedings of the 1990 ACM SIGMOD Conference*, pp. 322-331, Atlantic City, NJ, 1990.

[Brin93] T. Brinkhoff, H.P. Kriegel and B. Seeger: Efficient processing of spatial joins using R-trees, *Proceedings of the 1993 ACM SIGMOD Conference*, pp. 237-246, Washington, DC, 1993.

[Gunt89] O. Gunther: The design of the Cell tree: an object-oriented index structure for geometric databases, *Proceedings of the 5th IEEE Conference on Data Engineering*, pp. 598-615, Los Angeles, CA,1989.

[Gutt84] A. Guttman: R-trees: a dynamic index structure for spatial searching, *Proceedings of the 1984 ACM SIGMOD Conference*, pp. 47-57, Boston, MA, 1984.

[Henr89] A. Henrich, H.W. Six and P. Widmayer: The LSD-tree: spatial access to multidimensional point and non-point objects, *Proceedings of the 15th VLDB Conference*, pp. 45-53, Amsterdam, Netherlands, 1989.

[Laur92] R. Laurini and D. Thompson: *Fundamentals of spatial information systems*, Academic Press, London, 1992.

[Kame93] I. Kamel and C. Faloutsos: On packing R-trees, *Proceedings of the 2nd Conference on Information and Knowledge Management (CIKM)*, Washington DC, 1993.

[Kame94] I. Kamel and C. Faloutsos: Hilbert R-tree: an improved R-tree using fractals, *Proceedings of the 20th VLDB Conference*, pp. 500-509, Santiago, Chile, 1994.

[Lome90] D. Lomet and B. Salzberg: The hB-tree: A multiattribute indexing method with good guaranteed performance, *ACM Transactions on Database Systems*, Vol.15, No.4, pp. 625-658, 1990.

[Niev84] J. Nievergelt, H. Hinterberger and K.C. Sevcik: The grid-file: an adaptable, symmetric multikey file structure, *ACM Transactions on Database Systems*, Vol.9, No.1, pp. 38-71, 1984.

[Oren86] J. Orenstein: Spatial query processing in an object-oriented database system, *Proceedings of the 1986 ACM SIGMOD Conference*, pp.326-336, Washington DC, 1986.

[Page93] B.U. Pagel, H.W. Six and H. Toben: The transformation technique for spatial objects revisited, *Proceedings of the 3rd International Symposium on Spatial Databases (SSD93)*, pp. 73-88, Singapore 1993.

[Rous85] N. Roussopoulos and D. Leifker: Direct spatial search on pictorial databases using packed R-trees, *Proceedings of the 1985 ACM SIGMOD Conference*, pp. 17-31, Austin, TX, 1985.

[Same90] H. Samet: *The design and analysis of spatial data structures*, Addison-Wesley, Reading, MA, 1990.

[Sell87] T. Sellis, N. Roussopoulos and C. Faloutsos: The R^+-tree: a dynamic index for multidimensional objects, *Proceedings of the 13th VLDB Conference*, pp. 507-518, Brighton, UK, 1987.

[Ston93] M. Stonebraker, J. Frew, K. Gardels and J. Meredith: The Sequoia 2000 storage benchmark, *Proceedings of the 1993 ACM SIGMOD Conference*, pp. 2-11, Washington, DC, 1993.

[TIGE94] TIGER/Line(R) Files, Technical Documentation / prepared by the Bureau of the Census, Washington DC, 1994.