Hilbert R-tree: An improved R-tree using fractals

Ibrahim Kamel and *Christos Faloutsos*[*]

Department of Computer Science and
Institute for Systems Research (ISR)
University of Maryland
College Park, MD 20742

February 22, 1994

Abstract

We propose a new R-tree structure that outperforms all the older ones. The heart of the idea is to facilitate the deferred splitting approach in R-trees. This is done by proposing an ordering on the R-tree nodes. This ordering has to be 'good', in the sense that it should group 'similar' data rectangles together, to minimize the area and perimeter of the resulting minimum bounding rectangles (MBRs).

Following [19] we have chosen the so-called '2D-c' method, which sorts rectangles according to the Hilbert value of the center of the rectangles. Given the ordering, every node has a well-defined set of sibling nodes; thus, we can use deferred splitting. By adjusting the split policy, the Hilbert R-tree can achieve as high utilization as desired. To the contrary, the R^*-tree has no control over the space utilization, typically achieving up to 70%. We designed the manipulation algorithms in detail, and we did a full implementation of the Hilbert R-tree. Our experiments show that the '2-to-3' split policy provides a compromise between the insertion complexity and the search cost, giving up to 28% savings over the $R^* - tree$ [3] on real data.

1 Introduction

One of the requirements for the database management systems (DBMSs) of the near future is the ability to handle spatial data [28]. Spatial data arise in many applications, including: Cartography [29]; Computer-Aided Design (CAD) [24] [14]; computer vision and robotics [2]; traditional databases, where a record with k attributes corresponds to a point in a k-d space; temporal

[*]This research was partially funded by the Institute for Systems Research (ISR), by the National Science Foundation under Grants IRI-9205273 and IRI-8958546 (PYI), with matching funds from EMPRESS Software Inc. and Thinking Machines Inc.

databases, where time can be considered as one more dimension [20]; scientific databases with spatial-temporal data, such as the ones in the 'Grand Challenge' applications [10], etc.

In the above applications, one of the most typical queries is the *range query*: Given a rectangle, retrieve all the elements that intersect it. A special case of the range query is the *point query* or *stabbing query*, where the query rectangle degenerates to a point.

We focus on the R-tree [15] family of methods, which contains some of the most efficient methods that support range queries. The advantage of our method (and the rest of the R-tree-based methods) over the methods that use linear quad-trees and z-ordering is that R-trees treat the data objects as a whole, while quad-tree based methods typically divide objects into quad-tree blocks, increasing the number of items to be stored.

The most successful variant of R-trees seems to be the R^*-tree [3]. One of its main contributions is the idea of 'forced-reinsert' by deleting some rectangles from the overflowing node, and reinserting them.

The main idea in the present paper is to impose an ordering on the data rectangles. The consequences are important: using this ordering, each R-tree node has a well defined set of siblings; thus, we can use the algorithms for deferred splitting. By adjusting the split policy (2-to-3 or 3-to-4 etc) we can drive the utilization as close to 100% as desirable. Notice that the R^*-tree does not have control over the utilization, typically achieving an average of $\approx$70%.

The only requirement for the ordering is that it has to be 'good', that is, it should lead to small R-tree nodes.

The paper is organized as follows. Section 2 gives a brief description of the R-tree and its variants. Section 3 describes the Hilbert R-tree. Section 4 presents our experimental results that compare the Hilbert R-tree with other R-tree variants. Section 5 gives the conclusions and directions for future research.

2 Survey

Several spatial access methods have been proposed. A recent survey can be found in [26]. These methods fall in the following broad classes: methods that transform rectangles into points in a higher dimensionality space [16, 8]; methods that use linear quadtrees [9] [1] or, equivalently, the z-ordering [23] or other space filling curves [7] [18]; and finally, methods based on trees (R-tree [15], k-d-trees [4], k-d-B-trees [25], hB-trees [21], cell-trees [13] e.t.c.)

One of the most promising approaches in the last class is the R-tree [15]: Compared to the transformation methods, R-trees work on the native space, which has lower dimensionality; compared to the linear quadtrees, the R-trees do not need to divide the spatial objects into (several) pieces (quadtree blocks). The R-tree is the extension of the B-tree for multidimensional objects. A geometric object is represented by its minimum bounding rectangle (MBR). Non-leaf nodes contain entries of the form (R,ptr) where ptr is a pointer to a child node in the R-tree; R is the MBR that covers all rectangles in the child node. Leaf nodes contain entries of the form (obj-id, R) where obj-id is a pointer to the object description, and R is the MBR of the object. The main innovation in the R-tree is that father nodes are allowed to overlap. This way, the R-tree can guarantee at least 50% space utilization and remain balanced.

Guttman proposed three splitting algorithms, the *linear split*, the *quadratic split* and the *exponential split*. Their names come from their complexity; among the three, the quadratic split algorithm is the one that achieves the best trade-off between splitting time and search performance.

Subsequent work on R-trees includes the work by Greene [11], the R^+-tree [27], R-trees using Minimum Bounding Polygons [17], and finally, the R^*-tree [3], which seems to have the best performance among the R-tree variants. The main idea in the R^*-tree is the concept of *forced re-insert*. When a node overflows, some of its children are carefully chosen; they are deleted and re-inserted, usually resulting in a R-tree with better structure.

3 Hilbert R-trees

In this section we introduce the Hilbert R-tree and discuss algorithms for searching, insertion, deletion, and overflow handling. The performance of the R-trees depends on how good is the algorithm that cluster the data rectangles to a node. We propose to use space filling curves (or fractals), and specifically, the Hilbert curve to impose a linear ordering on the data rectangles.

A space filling curve visits all the points in a k-dimensional grid exactly once and never crosses itself. The Z-order (or Morton key order, or bit-interleaving, or Peano curve), the Hilbert curve, and the Gray-code curve [6] are examples of space filling curves. In [7], it was shown experimentally that the Hilbert curve achieves the best clustering among the three above methods.

Next we provide a brief introduction to the Hilbert curve: The basic Hilbert curve on a 2x2 grid, denoted by H_1, is shown in Figure 1. To derive a curve of order i, each vertex of the basic curve is replaced by the curve of order $i-1$, which may be appropriately rotated and/or reflected. Figure 1 also shows the Hilbert curves of order 2 and 3. When the order of the curve tends to infinity, the resulting curve is a *fractal*, with a fractal dimension of 2 [22]. The Hilbert curve can be generalized for higher dimensionalities. Algorithms to draw the two-dimensional curve of a given order, can be found in [12], [18]. An algorithm for higher dimensionalities is in [5].

The path of a space filling curve imposes a linear ordering on the grid points. Figure 1 shows

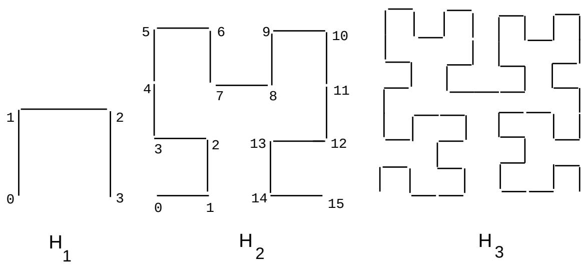

Figure 1: Hilbert Curves of order 1, 2 and 3

one such ordering for a 4×4 grid (see curve H_2). For example the point (0,0) on the H_2 curve has a Hilbert value of 0, while the point (1,1) has a Hilbert value of 2. The Hilbert value of a rectangle needs to be defined. Following the experiments in [19], a good choice is the following:

Definition 1 : *The Hilbert value of a rectangle is defined as the Hilbert value of its center.*

After this preliminary material, we are in a position now to describe the proposed methods.

3.1 Description

The main idea is to create a tree structure that can

- behave like an R-tree on search.
- support deferred splitting on insertion, using the Hilbert value of the inserted data rectangle as the primary key.

These goals can be achieved as follows: for every node n of our tree, we store (a) its MBR, and (b) the *Largest Hilbert Value (LHV)* of the data rectangles that belong to the subtree with root n.

Specifically, the Hilbert R-tree has the following structure. A leaf node contains at most C_l entries each of the form

$$(R, \; obj\_id)$$

where C_l is the capacity of the leaf, R is the MBR of the real object (x_{low} , x_{high} , y_{low} , y_{high}) and $obj - id$ is a pointer to the object description record. The main difference with R- and R*-trees is that nonleaf nodes also contain information about the LHVs. Thus, a non-leaf node in the Hilbert R-tree contains at most C_n entries of the form

$$(R, \; ptr, \; LHV)$$

where C_n is the capacity of a non-leaf node, R is the MBR that encloses all the children of that node, ptr is a pointer to the child node, and LHV is the largest Hilbert value among the *data* rectangles enclosed by R. Notice that we *never* calculate or use the Hilbert values of the MBRs. Figure 2 illustrates some rectangles, organized in a Hilbert R-tree. The Hilbert values of the centers are the numbers by the 'x' symbols (shown only for the parent node 'II'). The LHV's are in [brackets]. Figure 3 shows how is the tree of Figure 2 stored on the disk; the contents of the parent node 'II' are shown in more detail. Every data rectangle in node 'I' has Hilbert value $\leq$33; everything in node 'II' has Hilbert value greater than 33 and $\leq$107 etc.

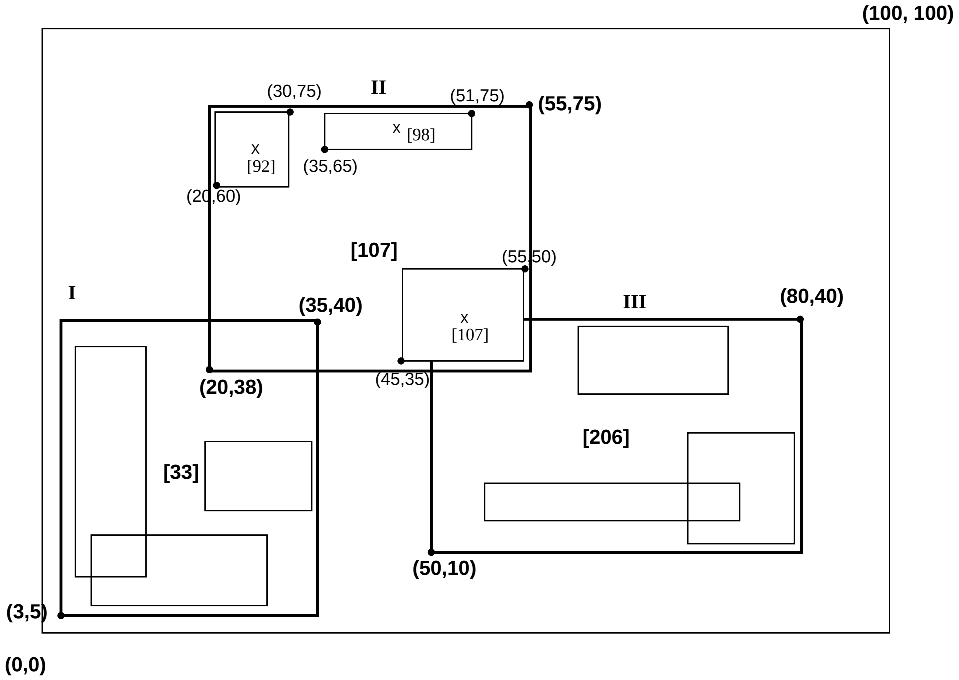

Figure 2: *Data rectangles organized in a Hilbert R-tree*

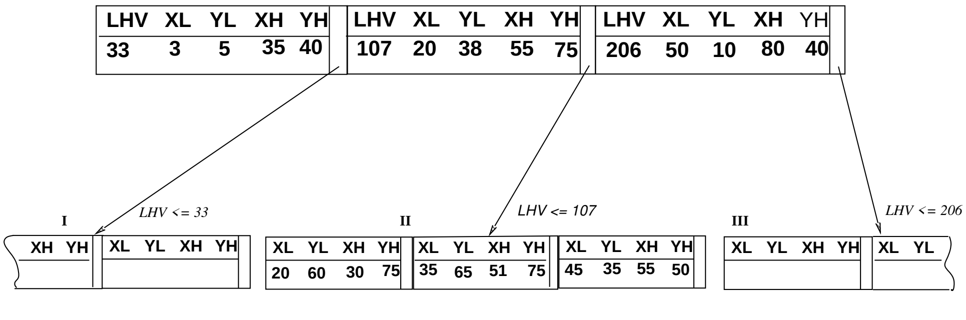

Figure 3: *The file structure for the previous Hilbert R-tree*

Before we continue, we list some definitions. A plain R-tree splits a node on overflow, turning 1 node to 2. We call this policy a *1-to-2* splitting policy. We propose to defer the split, waiting until they turn 2 nodes into 3. We refer to it as the 2-to-3 splitting policy. In general, we can have an s-to-(s+1) splitting policy; we refer to s as the *order of the splitting policy.* To implement the order-s splitting policy, the overflowing node tries to push some of its entries to one of its $s-1$ siblings; if all of them are full, then we have an s-to-(s+1) split. We refer to these $s-1$ siblings as the *cooperating siblings* of a given node.

Next, we will describe in detail the algorithms for searching, insertion, and overflow handling.

3.2 Searching

The searching algorithm is similar to the one used in other R-tree variants. Starting from the root it descends the tree examining all nodes that intersect the query rectangle. At the leaf level it reports all entries that intersect the query window w as qualified data items.

Algorithm Search(node Root, rect w):

S1. *Search nonleaf nodes:*

invoke Search for every entry whose MBR intersects the query window w.

S2. *Search leaf nodes:*

Report all the entries that intersect the query window w as candidate.

3.3 Insertion

To insert a new rectangle r in the Hilbert R-tree, the Hilbert value h of the center of the new rectangle is used as a key. In each level we choose the node with minimum LHV among the siblings. When a leaf node is reached the rectangle r is inserted in its correct order according to h. After a new rectangle is inserted in a leaf node N, **AdjustTree** is called to fix the MBR and LHV values in upper level nodes.

Algorithm Insert(node Root, rect r):

/* inserts a new rectangle r in the Hilbert R-tree. h is the Hilbert value of the rectangle. */

I1. *Find the appropriate leaf node:*

Invoke **ChooseLeaf(r, h)** to select a leaf node L in which to place r.

I2. *Insert r in a leaf node L:*

if L has an empty slot, insert r in L in the appropriate place according to the Hilbert order and return.

if L is full, invoke **HandleOverflow(L,r)**, which will return new leaf if split was inevitable.

I3. *Propagate changes upward:*

form a set $\mathcal{S}$ that contains L, its cooperating siblings and the new leaf (if any).

invoke **AdjustTree($\mathcal{S}$)**

I4. *Grow tree taller:*

if node split propagation caused the root to split, create a new root whose children are the two resulting nodes.

Algorithm ChooseLeaf(rect r, int h):

/* Returns the leaf node in which to place a new rectangle r. */

C1. *Initialize:*

Set N to be the root node.

C2. *Leaf check:*

if N is a leaf, return N.

C3. *Choose subtree:*

if N is a non-leaf node, choose the entry (R, ptr, LHV) with the minimum LHV value greater than h.

C4. *Descend until a leaf is reached:*

set N to the node pointed by ptr and repeat from C2.

Algorithm AdjustTree(set $\mathcal{S}$):

/* $\mathcal{S}$ is a set of nodes that contains the node being updated, its cooperating siblings (if overflow has occurred) and newly created node NN (if split has occurred).

The routine ascends from leaf level towards the root, adjusting MBR and LHV of nodes that cover the nodes in $\mathcal{S}$.

siblings. It propagates splits (if any). */

A1. if reached root level stop.

A2. *Propagate node split upward*

let N_p be the parent node of N.

if N has been split, let NN be the new node.

insert NN in N_p in the correct order according to its Hilbert value if there is room. Otherwise, invoke **HandleOverflow(**N_p**, NN)**.

if N_p is split, let PP be the new node.

A3. *adjust the MBR's and LHV's in the parent level:*

let $\mathcal{P}$ be the set of parent nodes for the nodes in $\mathcal{S}$.

Adjust the corresponding MBR's and LHV's appropriately of the nodes in $\mathcal{P}$.

A4. *Move up to next level:*

Let $\mathcal{S}$ become the set of parent nodes $\mathcal{P}$, with

NN = PP, if N_p was split.

repeat from A1.

3.4 Deletion

In Hilbert R-tree we do NOT need to re-insert orphaned nodes, whenever a father node underflows. Instead, we borrow keys from the siblings or we merge an underflowing node with its siblings. We are able to do so, because the nodes have a clear ordering (Largest Hilbert Value LHV); in contrast, in R-trees there is no such concept of sibling node. Notice that, for deletion, we need s cooperating siblings while for insertion we need $s-1$.

Algorithm Delete(r):

D1. *Find the host leaf:*

Perform an exact match search to find the leaf node L that contain r.

D2. *Delete* r :

Remove r from node L.

D3. if L underflows

borrow some entries from s cooperating siblings.

if all the siblings are ready to underflow,

merge $s+1$ to s nodes,

adjust the resulting nodes.

D4. *adjust MBR and LHV in parent levels.*

form a set $\mathcal{S}$ that contains L and its cooperating siblings (if underflow has occurred).

invoke **AdjustTree**($\mathcal{S}$).

3.5 Overflow handling

The overflow handling algorithm in the Hilbert R-tree treats the overflowing nodes either by moving some of the entries to one of the $s-1$ cooperating siblings or splitting s nodes to $s+1$ nodes.

Algorithm HandleOverflow(node N, rect r):

/* return the new node if a split occurred. */

H1. let $\mathcal{E}$ be a set that contains all the entries from N and its $s-1$ cooperating siblings.

H2. add r to $\mathcal{E}$.

H3. if at least one of the $s-1$ cooperating siblings is not full,

distribute $\mathcal{E}$ evenly among the s nodes according to the Hilbert value.

H4. if all the s cooperating siblings are full,

create a new node NN and

distribute $\mathcal{E}$ evenly among the $s+1$ nodes according to the Hilbert value.

return NN.

4 Experimental results

To assess the merit of our proposed Hilbert R-tree, we implemented it and ran experiments on a two dimensional space. The method was implemented in C, under UNIX. We compared our methods against the quadratic-split R-tree, and the $R^* - tree$. Since the CPU time required to process the node is negligible, we based our comparison on the number of nodes (=pages) retrieved by range queries.

Without loss of generality, the address space was normalized to the unit square. There are several factors that affect the search time; we studied the following ones:

Data items: points and/or rectangles and/or line segments (represented by their MBR)

File size: ranged from 10,000 - 100,000 records

Query area $Q_{area} = q_x \times q_y$**:** ranged from 0 - 0.3 of the area of the address space

Another important factor, which is derived from N and the average area a of the data rectangles, is the 'data density' d (or 'cover quotient') of the data rectangles. This is the sum of the areas of the data rectangles in the unit square, or equivalently, the average number of rectangles that cover a randomly selected point. Mathematically: $d = N \times a$. For the selected values of N and a, the data density ranges from 0.25 - 2.0.

To compare the performance of our proposed structures we used 5 data files that contained different types of data: points, rectangles, lines, or mixed. Specifically, we used:

A) Real Data: we used real data from the TIGER system of the U.S. Bureau of Census. An important observation is that the data in the TIGER datasets follow a highly *skewed* distribution.

- **'MGCounty' :** This file consists of 39717 line segments, representing the roads of Montgomery county in Maryland. Using the minimum bounding rectangles of the segments, we obtained 39717 rectangles, with data density $d = 0.35$. We refer to this dataset as the '*MGCounty*' dataset.

- **'LBeach' :** It consists of 53145 line segments, representing the roads of Long Beach, California. The data density of the MBRs that cover these line segments is $d = 0.15$. We refer to this dataset as the '*LBeach*' dataset.

B) Synthetic Data: The reason for using synthetic data is that we can control the parameters (data density, number of rectangles, ratio of points to rectangles etc.).

- **'Points' :** This file contains 75,000 uniformly distributed points.

- **'Rects' :** This file contains 100,000 rectangles, no points. The centers of the rectangles are uniformly distributed in the unit square. The data density is $d = 1.0$

- **'Mix' :** This file contains a mix of points and rectangles; specifically 50,000 points and 10,000 rectangles; the data density is $d = 0.029$.

The query rectangles were squares with side q_s; their centers were uniformly distributed in the unit square. For each experiment, 200 randomly generated queries were asked and the results were averaged. The standard deviation was very small and is not even plotted in our graphs. The page size used is 1KB.

We compare the Hilbert R-tree against the original R-tree (quadratic split) and the R^*-tree. Next we present experiments that (a) compare our method against other R-tree variants (b) show the effect of the different split policies on the performance of the proposed method and (c) evaluate the insertion cost.

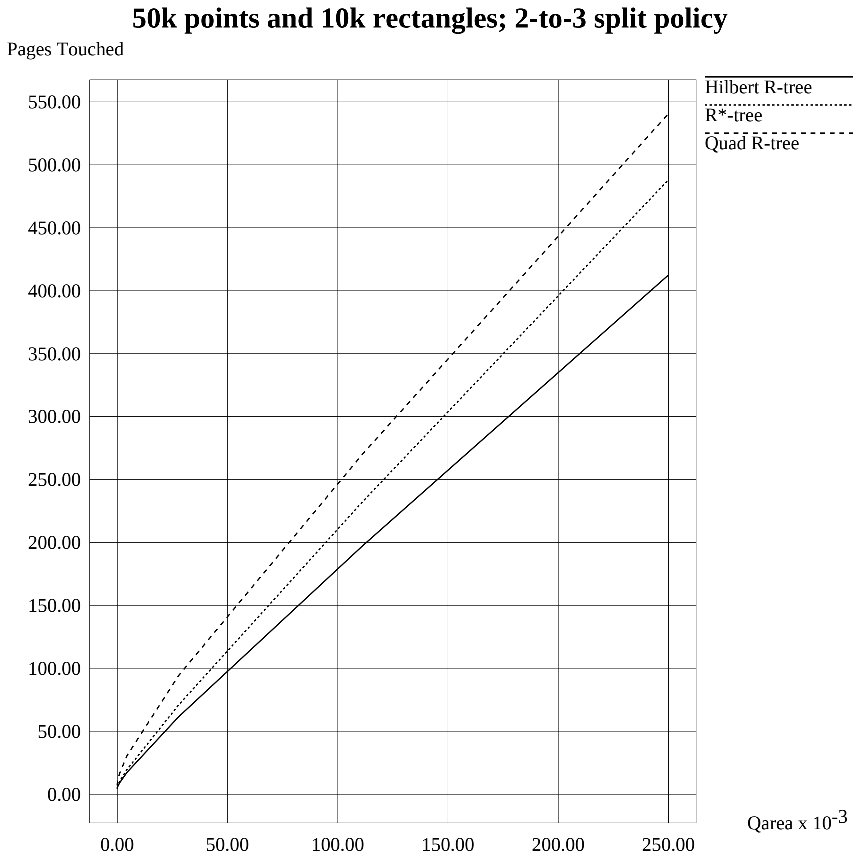

Figure 4: *points and rectangles ('Mix' dataset); disk accesses vs. query area*

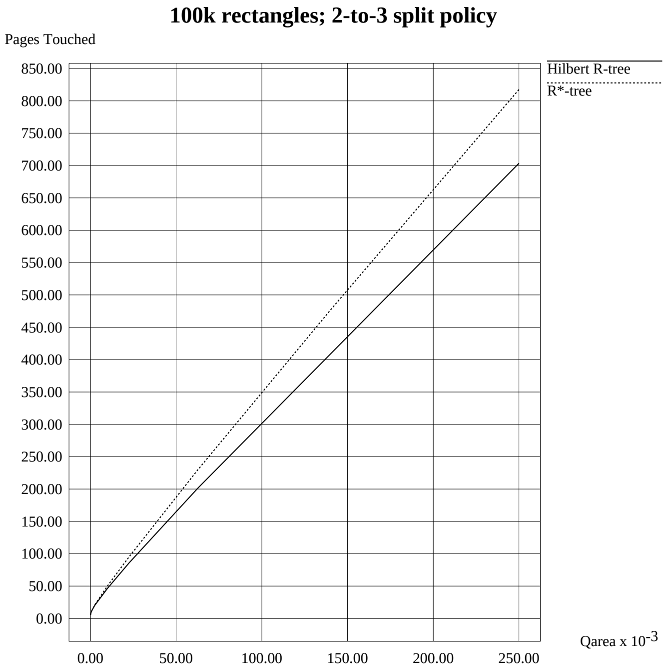

Figure 5: *Rectangles Only ('Rects' dataset); disk accesses vs. query area*

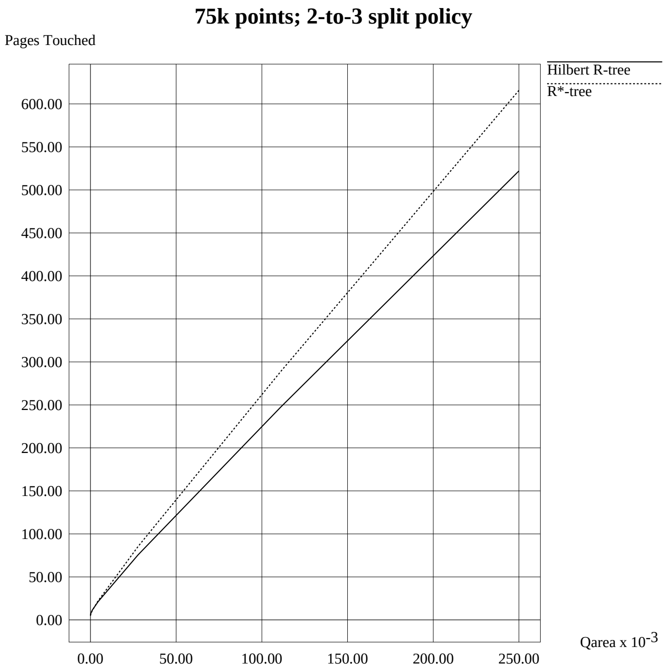

Figure 6: *Points Only ('Points' dataset); disk accesses vs. query area*

4.1 Comparison of the Hilbert R-tree vs. other R-tree variants

In this section we show the performance superiority of our Hilbert R-tree over the $R^* - tree$, which is the most successful variant of the R-tree. We present experiments with all five datasets, namely: 'Mix', 'Rects', 'Points', 'MGCounty', and 'LBeach' (see Figures 4 - 6, respectively). In all these experiments, we used the '2-to-3' split policy for the Hilbert R-tree.

In all the experiment the Hilbert R-tree is the clear winner, achieving up to 28% savings in response time over the next best contender (the $R^* - tree$). This maximum gain is achieved for the 'MGCounty' dataset (Figure 7). It is interesting to notice that the performance gap is larger for the real data, whose main difference from the synthetic one is that it is skewed, as opposed to uniform. Thus, we can conjecture that the skeweness of the data favors the Hilbert R-tree.

Figure 4 also plots the results for the quadratic-split R-tree, which, as expected, is outperformed by the $R^* - tree$. In the rest of the figures, we omit the quadratic-split R-tree, because it was consistently outperformed by $R^* - tree$.

4.2 The effect of the split policy on the performance

Figure 9 shows the response time as a function of the query size for the 1-to-2, 2-to-3, 3-to-4 and 4-to-5 split policies. The corresponding space utilization was 65.5%, 82.2%, 89.1% and 92.3% respectively. For comparison, we also plot the response times of the $R^* - tree$. As expected, the response time for the range queries improves with the average node utilization. However, there seems to be a point of diminishing returns as s increases. For this reason, we recommend the '2-to-3' splitting policy (s=2), which strikes a balance between insertion speed (which deteriorates with s) and search speed, which improves with s.

4.3 Insertion cost

The higher space utilization in the Hilbert R-tree comes at the expense of higher insertion cost. As we employ higher split policy the number of cooperating siblings need to be inspected at overflow increases. We see that '2-to-3' policy is a good compromise between the performance and the insertion cost. In this section we compare the insertion cost of the Hilbert R-tree '2-to-3' split with the insertion cost in the $R^* - tree$. Also, show the effect of the split policy on the insertion cost. The cost is measured by the number of disk accesses per insertion.

Table 1 shows the insertion cost of the Hilbert R-tree and the $R^* - tree$ for the five different datasets. The main observation here is that there is no clear winner in the insertion cost.

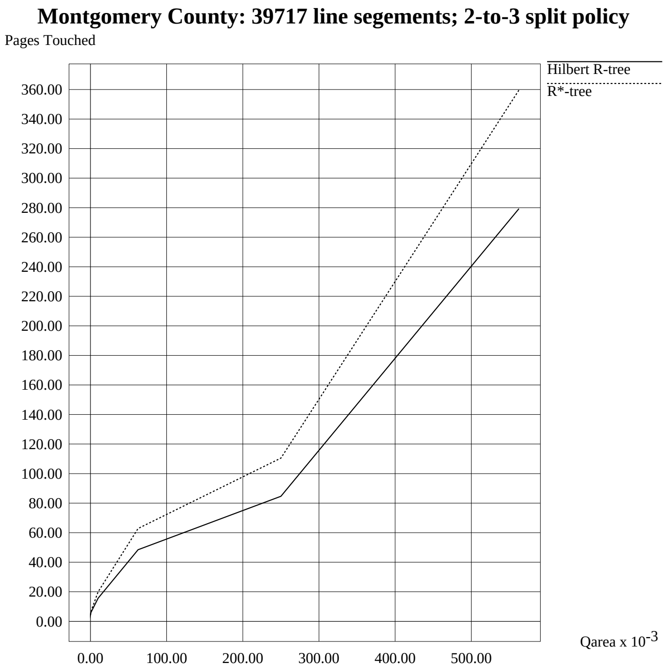

Figure 7: *Montgomery County dataset; disk accesses vs. query area*

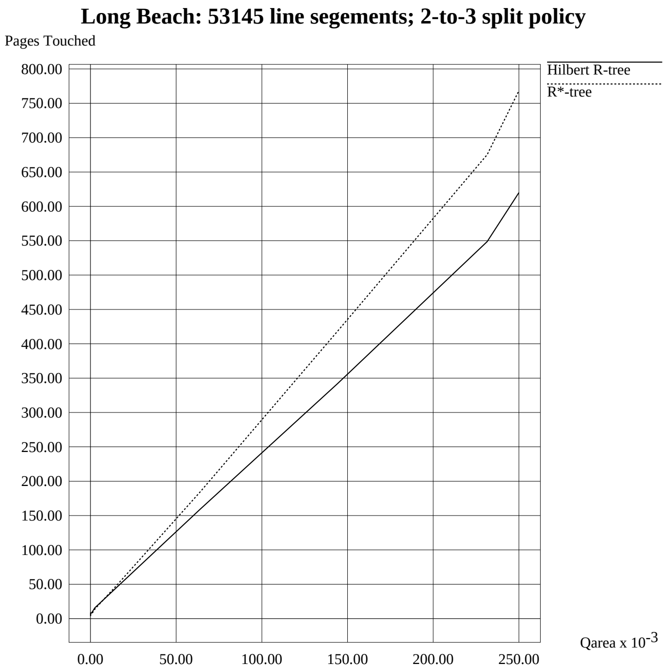

Figure 8: *Long Beach dataset; disk accesses vs. query area*

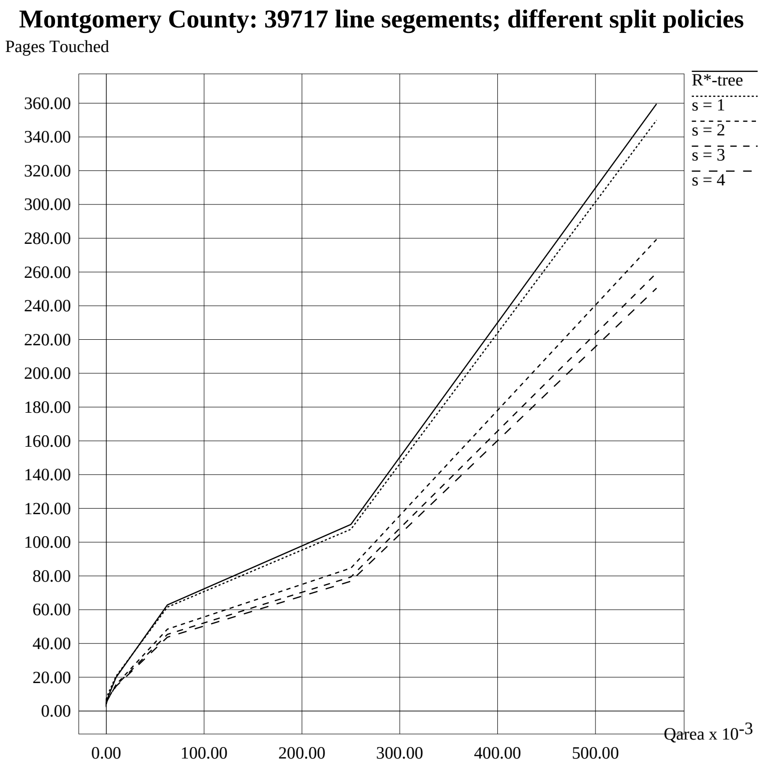

Figure 9: *The effect of the split policy; disk accesses vs. query area*

Table 2 shows the effect of increasing the split policy in the Hilbert R-tree on the insertion cost for $MGCounty$ dataset. As expected, the insertion cost increases with the order s of the split policy.

5 Conclusions

In this paper we designed and implemented a superior R-tree variant, which outperforms all the previous R-tree methods. The major idea is to introduce a 'good' ordering among rectangles. By simply defining an ordering, the R-tree structure is amenable to deferred splitting, which can make the utilization approach the 100% mark as closely as we want. Better packing results in a shallower tree and a higher fanout. If the ordering happens to be 'good', that is, to group similar rectangles together, then the R-tree will in addition have nodes with small MBRs, and eventually, fast response times.

Based on this idea, we designed in detail and implemented the Hilbert R-tree, a dynamic tree structure that is capable of handling insertions and deletions. Experiments on real and synthetic data showed that the proposed Hilbert R-tree with the '2-to-3' splitting policy consistently

dataset	(disk accesses)/insertion	
	Hilbert R-tree (2-to-3 split)	$R^* - tree$
MGCounty	3.55	3.10
LBeach	3.56	4.01
Points	3.66	4.06
Rects	3.95	4.07
Mix	3.47	3.39

Table 1: Comparison between insertion cost in Hilbert R-tree '2-to-3' split and $R^* - tree$; disk accesses per insertion

split policy	(disk accesses)/insertion
1-to-2	3.23
2-to-3	3.55
3-to-4	4.09
4-to-5	4.72

Table 2: The effect of the split policy on the insertion cost; MGCounty dataset

outperforms all the R-tree methods, with up to 28% savings over the best competitor (the R^*-tree).

Future work could focus on the analysis of Hilbert R-trees, providing analytical formulas that predict the response time as a function of the characteristics of the data rectangles (count, data density etc).

References

[1] Walid G. Aref and Hanan Samet. Optimization strategies for spatial query processing. *Proc. of VLDB (Very Large Data Bases)*, pages 81–90, September 1991.

[2] D. Ballard and C. Brown. *Computer Vision.* Prentice Hall, 1982.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: an efficient and robust access method for points and rectangles. *ACM SIGMOD*, pages 322–331, May 1990.

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

[5] T. Bially. Space-filling curves: Their generation and their application to bandwidth reduction. *IEEE Trans. on Information Theory*, IT-15(6):658–664, November 1969.

[6] C. Faloutsos. Gray codes for partial match and range queries. *IEEE Trans. on Software Engineering*, 14(10):1381–1393, October 1988. early version available as UMIACS-TR-87-4, also CS-TR-1796.

[7] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. *Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 247–252, March 1989. also available as UMIACS-TR-89-47 and CS-TR-2242.

[8] Michael Freeston. The bang file: a new kind of grid file. *Proc. of ACM SIGMOD*, pages 260–269, May 1987.

[9] I. Gargantini. An effective way to represent quadtrees. *Comm. of ACM (CACM)*, 25(12):905–910, December 1982.

[10] Grand challenges: High performance computing and communications, 1992. The FY 1992 U.S. Research and Development Program.

[11] D. Greene. An implementation and performance analysis of spatial data access methods. *Proc. of Data Engineering*, pages 606–615, 1989.

[12] J.G. Griffiths. An algorithm for displaying a class of space-filling curves. *Software-Practice and Experience*, 16(5):403–411, May 1986.

[13] O. Gunther. The cell tree: an index for geometric data. Memorandum No. UCB/ERL M86/89, Univ. of California, Berkeley, December 1986.

[14] A. Guttman. *New Features for Relational Database Systems to Support CAD Applications.* PhD thesis, University of California, Berkeley, June 1984.

[15] A. Guttman. R-trees: a dynamic index structure for spatial searching. *Proc. ACM SIGMOD*, pages 47–57, June 1984.

[16] K. Hinrichs and J. Nievergelt. The grid file: a data structure to support proximity queries on spatial objects. *Proc. of the WG'83 (Intern. Workshop on Graph Theoretic Concepts in Computer Science)*, pages 100–113, 1983.

[17] H. V. Jagadish. Spatial search with polyhedra. *Proc. Sixth IEEE Int'l Conf. on Data Engineering*, February 1990.

[18] H.V. Jagadish. Linear clustering of objects with multiple attributes. *ACM SIGMOD Conf.*, pages 332–342, May 1990.

[19] I. Kamel and C. Faloutsos. On packing r-trees. In *Proc. 2nd International Conference on Information and Knowledge Management(CIKM-93)*, pages 490–499, Arlington, VA, November 1993.

[20] Curtis P. Kolovson and Michael Stonebraker. Segment indexes: Dynamic indexing techniques for multi-dimensional interval data. *Proc. ACM SIGMOD*, pages 138–147, May 1991.

[21] David B. Lomet and Betty Salzberg. The hb-tree: a multiattribute indexing method with good guaranteed performance. *ACM TODS*, 15(4):625–658, December 1990.

[22] B. Mandelbrot. *Fractal Geometry of Nature.* W.H. Freeman, New York, 1977.

[23] J. Orenstein. Spatial query processing in an object-oriented database system. *Proc. ACM SIGMOD*, pages 326–336, May 1986.

[24] J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott, and G. S. Taylor. Magic: a vlsi layout system. In *21st Design Automation Conference*, pages 152 – 159, Alburquerque, NM, June 1984.

[25] J.T. Robinson. The k-d-b-tree: a search structure for large multidimensional dynamic indexes. *Proc. ACM SIGMOD*, pages 10–18, 1981.

[26] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1989.

[27] T. Sellis, N. Roussopoulos, and C. Faloutsos. The r+ tree: a dynamic index for multi-dimensional objects. In *Proc. 13th International Conference on VLDB*, pages 507–518, England,, September 1987. also available as SRC-TR-87-32, UMIACS-TR-87-3, CS-TR-1795.

[28] Avi Silberschatz, Michael Stonebraker, and Jeff Ullman. Database systems: Achievements and opportunities. *Comm. of ACM (CACM)*, 34(10):110–120, October 1991.

[29] M. White. *N-Trees: Large Ordered Indexes for Multi-Dimensional Space.* Application Mathematics Research Staff, Statistical Research Division, U.S. Bureau of the Census, December 1981.