On Packing R-trees

Ibrahim Kamel and *Christos Faloutsos*[*]

Department of CS
University of Maryland
College Park, MD 20742

Abstract

We propose new R-tree packing techniques for static databases. Given a collection of rectangles, we sort them and we build the R-tree bottom-up. There are several ways to sort the rectangles; the innovation of this work is the use of fractals, and specifically the hilbert curve, to achieve better ordering of the rectangles and eventually better packing. We proposed and implemented several variations and performed experiments on synthetic, as well as real data (TIGER files from the U.S. Bureau of Census). The winning variation ('2D-c') was the one that sorts the rectangles according to the hilbert value of the center. This variation consistently outperforms the packing method of Roussopoulos and Leifker [24], as well as other R-tree variants. The performance gain of the our method seems to increase with the skeweness of the data distribution; specifically, on the (highly skewed) TIGER dataset, it achieves up to 58% improvement in response time over the older packing algorithm and 36% over the best known R-tree variant. We also, introduce an analytical formula to compute the average response time of a range query as a function of the geometric characteristics of the R-tree.

1 Introduction

One of the requirements for the database management systems (DBMSs) of the near future is the ability to handle spatial data. Spatial data arise in many applications, including: Cartography [27]; Computer-Aided Design (CAD) [22] [12]; computer vision and robotics [2]; traditional databases, where a record with k attributes corresponds to a point in a k-d space; temporal databases, where time can be considered as one more dimension [18]; scientific databases, with spatial-temporal data, etc.

[*]Currently on sabbatical at IBM Almaden Research Center. This research was partially funded by the Systems Research Center (SRC) at the University of Maryland, and by the National Science Foundation under Grant IRI-8958546 (PYI), with matching funds from EMPRESS Software Inc. and Thinking Machines Inc.

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 R-trees, which is one of the most efficient methods that support range queries. The original R-tree [13] and almost all of its variants are designed for a dynamic environment, being able to handle insertions and deletions. Operating in this mode, the R-tree guarantees that the space utilization is at least 50%; experimental results [3] showed that the average utilization is $\approx 70\%$. However, for a static set of data rectangles, we should be able to do better. Improving the space utilization through careful packing will have a two benefits: Not only does it reduce the space overhead of the R-tree index, but it can also offer better response time, because the R-tree will have a higher fanout and it will possibly be shorter.

Static data appear in several applications. For example, in cartographic databases, insertions and deletions are rare; databases with census results are static; the same is true for databases that are published on CD-ROMs; databases with spatio/temporal meteorological and environmental data are seldom modified, too. The volume of data in these databases is expected to be enormous, in which case it is crucial to minimize the space overhead of the index.

In this paper we study the problem of how to create a compressed R-tree for static data. Our goal is to develop a packing method such that the resulting R-tree (a) will have 100% space utilization and (b) will respond to range queries at least as fast as the R-tree of any other known method, (e.g. , the $R^* - tree$ or the quadratic-split R-tree).

We design and study several heuristics to build the R-tree bottom-up. Most of these heuristics are based on space filling curves, and specifically, the Hilbert curve. We report experiments from 2-dimensional data, although our method can handle higher dimensionalities. The experimental results showed that, the most effective of our heuristics is the one that sorts the data rectangles according to the hilbert value of their centers ('2D-c' heuristic). This heuristic consistently gives significant saving in response time over all the known R-tree variants, namely, the quadratic-split R-tree, the $R^* - tree$, as well as the method proposed by Roussopoulos and Leifker [24], which is the only R-tree packing known up to now.

We also propose an analytical formula to compute the average response time of a range query by using the area and perimeter of the R-tree nodes. This formula works for *any* R-tree variation, either static or dynamic.

The paper is organized as follows. Section 2 gives first a brief description of the R-tree and its variants. Section 3 describes our proposed heuristics to build the compressed R-tree. In section 4, we introduce the analytical formula to compute the average response time for a given R-tree instance, given some information about the minimum bounding rectangles of its nodes. Section 5 presents our experimental results that compare the proposed schemes with others. Section 6 gives the conclusions and directions for future research.

2 Survey

Several spatial access methods have been proposed. A recent survey can be found in [25]. These methods fall in the following broad classes: methods that transform rectangles into points in a higher dimensionality space [15]; methods that use linear quadtrees [8] [1] or, equivalently, the z-ordering [21] or other space filling curves [7] [17]; and finally, methods based on trees (R-tree [13], k-d-trees [4], k-d-B-trees [23], hB-trees [19], cell-trees [11] e.t.c.)

One of the most promising approaches in the last class is the R-tree [13]. It 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 (ptr,R) 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.

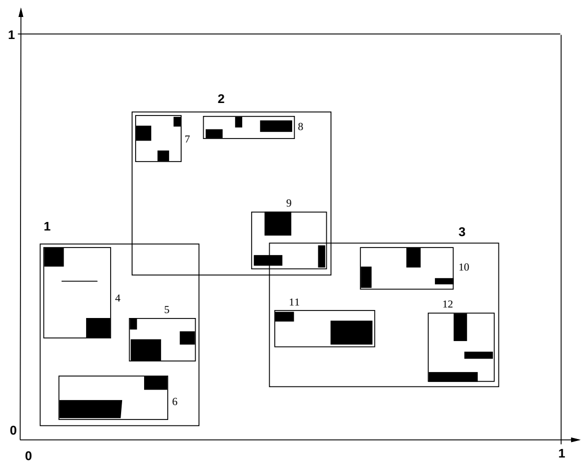

Figure 1: *Data (dark rectangles) organized in an R-tree with fanout=3*

Figure 1 illustrates data rectangles (in black), organized in an R-tree with fanout 3. Figure 2 shows the file structure for the same R-tree, where nodes correspond to disk pages. In the rest of this paper, the term 'node' and the term 'page' will be used interchangeably. 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

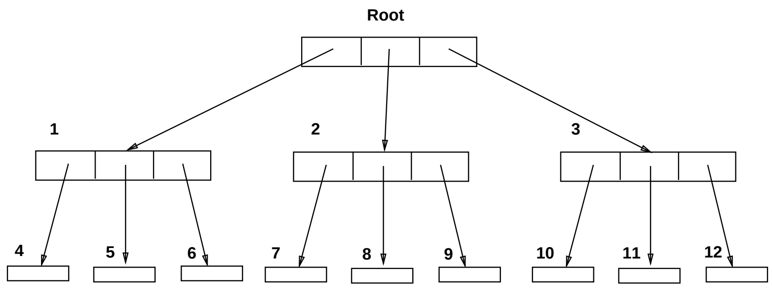

Figure 2: The file structure for the R-tree of the previous figure (fanout=3)

the best trade-off between splitting time and search performance.

Subsequent work on R-trees includes the work by Greene [9], the R^+-tree [26], R-trees using Minimum Bounding Polygons [16], 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*, which is analog to the deferred-splitting in B-trees. When a node overflows, some of its children are carefully chosen and they are deleted and re-inserted, usually resulting in a R-tree with better structure.

All the above R-tree variants support insertions and deletions, and are thus suitable for *dynamic* environments. For a static environment, the only R-tree packing scheme we are aware of is the method of Roussopoulos and Leifker [24]. They proposed a method to build a packed R-tree that achieves (almost) 100% space utilization. The idea is to sort the data on the x or y coordinate of one of the corners of the rectangles. The sorted list of rectangles is scanned; successive rectangles are assigned to the same R-tree leaf node, until this node is full; then, a new leaf node is created and the scanning of the sorted list continues. Thus, the nodes of the resulting R-tree will be fully packed, with the possible exception of the last node at each level. Thus, the utilization is $\approx 100\%$. Their experimental results on point data showed that their packed R-tree performs much better than the linear split R-tree for point queries. In our experiments (section 5), their packed R-tree outperformed the quadratic split R-tree and the $R^* - tree$ as well, for point queries on point data. However, it does not perform that well for region queries and/or rectangular data.

For the rest of this paper, we shall refer to this method by *lowx packed R-tree*. In our implementation of their method, we sort the rectangles according to their x value of the lower left corner ('$lowx$'). Sorting on any of the other three values gives similar results; thus our implementation does not impose an unfair disadvantage to the $lowx$ packed R-tree.

3 Proposed design

We assume that the data are static, or that the frequency of modification is low. Our goal is to design a simple heuristic to construct an R-tree with 100% space utilization, which, on the same time, will have as good response time as possible. The *lowx* packed R-tree [24] is a step towards this goal. However, it suffers from a subtle pitfall. Although it performs very well for point queries on point data, its performance degrades for larger queries. Figures 3 and 4 highlight the problem. Figure 4 shows the leaf nodes of the R-tree that the *lowx* packing method will create for the points of figure 3. The fact that the resulting father nodes cover little area explains why the *lowx* packed R-tree achieves excellent performance for point queries; the fact that the fathers have large perimeters, in conjunction with the upcoming Eq. 3 (section 4), explains the degradation of performance for region queries. Intuitively, the packing algorithm should ideally assign nearby points to the same leaf node; ignoring the y-coordinate, the *lowx* packed R-tree to tends violate this empirical rule.

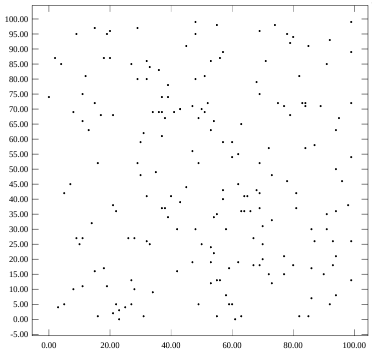

Figure 3: *200 points uniformly distributed*

In order to cluster the data in a better way than the *lowx* packed R-trees, we propose to use space filling curves (or fractals), and specifically, the Hilbert curve.

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 5. 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 5 also shows the Hilbert curves of order 2 and 3. When the order of the curve tends to

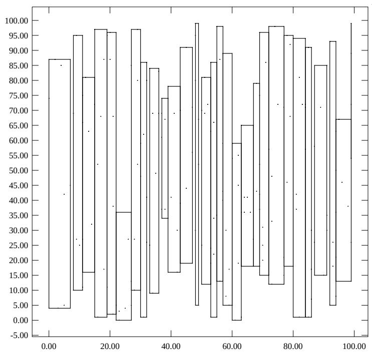

Figure 4: *MBR of nodes generated by the 'lowx packed R-tree' algorithm*

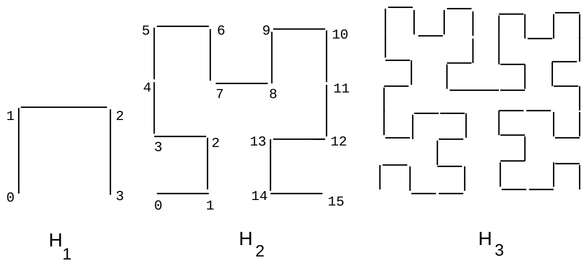

Figure 5: Hilbert Curves of order 1, 2 and 3

infinity, the resulting curve is a *fractal*, with a fractal dimension of 2 [20]. The Hilbert curve can be generalized for higher dimensionalities. Algorithms to draw the two-dimensional curve of a given order, can be found in [10], [17]. An algorithm for higher dimensionalities is in [5].

The path of a space filling curve imposes a linear ordering on the grid points, which may be calculated by starting at one end of the curve and following the path to the other end. Figure 5 shows 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.

After this preliminary material, we are in a position now to describe the proposed methods. Exploiting the superior clustering that the Hilbert curve can achieve, we impose a linear ordering on the data rectangles and then we traverse the sorted list, assigning each set of C rectangles to a node in the R-tree. The final result is that the set of data rectangles on the same node will be close to each other in the linear ordering, and most likely in the native space; thus the resulting R-tree nodes will have smaller areas. Figure 5 illustrates the intuitive reasons that our hilbert-based methods will result in good performance. The data are points (the same points with Figure 3 and Figure 4). We see that, by grouping the points according to their hilbert values, the MBRs of the resulting R-tree leaf nodes tend to be small *square*-like rectangles. This indicates that the nodes will likely have small area and small perimeter. Small area results in good performance for point queries; small area *and* small perimeter leads to good performance for larger queries. Eq. 3 confirms the above claims.

Algorithm Hilbert-Pack: packs rectangles into an R-tree
Step 1. Calculate the hilbert value for each data rectangle
Step 2. Sort data rectangles on ascending hilbert values
Step 3. */* Create leaf nodes (level l=0) */*
    While (there are more rectangles)
        generate a new R-tree node
        assign the next C rectangles to this node
Step 4. */* Create nodes at higher level ($l + 1$) */*
    While (there are > 1 nodes at level l)
        sort nodes at level $l \geq 0$ on ascending *creation* time
        repeat Step 3

Figure 6: Pseudo-code of the packing algorithm

We studied several methods to sort the data rectangles. All of them use the same algorithm (see Figure 6) to build the R-tree. The only point that the proposed hilbert-based methods distinguish themselves from each other is on the way they compute the hilbert value of a rectangle. We examine the following alternatives:

4d Hilbert through corners ("4D-xy"): Each data rectangle is mapped to a point in four dimensional space formed by the lower left corner and the upper right corner namely (lowx, lowy, highx, highy). The hilbert value of this 4-dimensional point is the hilbert value of the rectangle.

4-d Hilbert through center and diameter ("4D-cd"): Each data rectangle is mapped to the 4-dimensional point (c_x, c_y, d_x, d_y). where c_x, c_y are the coordinates of the center of the rectangle and d_x, d_y the 'diameters' or sides of the rectangle. As in 4D-xy, the hilbert value of this 4-dimensional point is the hilbert value of the rectangle.

2-d Hilbert through Centers Only ("2D-c"): Each data rectangle is represented by its *center only*; the hilbert value of the center is the hilbert value of the rectangle.

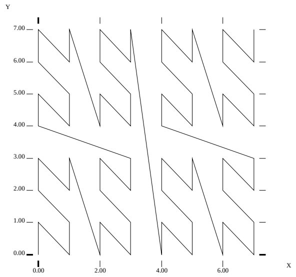

Figure 7: Peano (or z-order) curve of order 3

Method name	Description
2D-c	*sorts on the 2d-hilbert value of the centers* (c_x, c_y)
4D-xy	*sorts on the 4-d hilbert value of the two corners, i.e.,* $(low_x, low_y, high_x, high_y)$
4D-cd	*sorts on 4-d hilbert value of the center and diameters, i.e.,* (c_x, c_y, d_x, d_y)
2Dz-c	*sorts on the z-value of the center* (c_x, c_y)
lowx packed R-tree [24]	sorts on the x coordinate of the lower left corner
linear-split R-tree [13]	Guttman's R-tree, with linear split
quadratic-split R-tree [13]	Guttman's R-tree with quadratic split
R^*-tree [3]	R-tree variant, better packing, forced reinsert

Table 1: List of methods - the proposed ones are in *italics*

For the sake of comparison, we also examined a method that uses the Peano curve, or 'z-ordering', despite the fact that the z-ordering achieves inferior clustering compared to the hilbert curve. The z-value of a point is computed by bit-interleaving the binary representation of its x and y coordinates. For example, in Figure 7, the point (0,0) has a z-value of 0, while the point (1,3) has a z-value of 7.

Z-order through Centers only ("2Dz-c"): The value of the rectangle is the z-value of its center.

Table 1 gives a list of the methods we compared, along with a brief description of each. The new methods are in italics; R-tree methods for static environments are above the double horizontal line; the rest can be applied for dynamic environments, as well.

Symbols	Definitions
p	page size, in bytes
C	page capacity (max. number of rectangles per page)
$P(q_x, q_y)$	avg. pages retrieved by a $q_x \times q_y$ query
N_d	number of data rectangles
N	number of tree nodes
d	density of data
n_i	node i in the R-tree
$n_{i,x}$	length of node i in x direction
$n_{i,y}$	length of node i in y direction
L_x	sum of x-sides of all nodes in the tree
L_y	sum of y-sides of all nodes in the tree
$TotalArea$	sum of areas of all nodes in the tree
q_x	length of the query in x direction
q_y	length of the query in y direction

Table 2: Summary of Symbols and Definitions

4 Analytical formula for the response time

In this section we introduce an analytical formula to evaluate the average response time for a query of size $q_x \times q_y$ as a function of the geometric characteristics of the R-tree. This means that once we built the R-tree we can estimate the average response time of the query $q_x \times q_y$ *without* generating random queries and computing the average and variance of their response times. In this discussion we assume that queries are rectangles uniformly distributed over the unit square address space. Without loss of generality we consider a 2-dimensional space. The same idea can be generalized to higher dimensionalities.

The response time of a range query is mainly affected by the time required to retrieve the nodes touched by the query plus the time required by the CPU to process the nodes. Since the CPU is much faster than the disk, we assume that the CPU time is negligible (=0) compared to the time required by a disk to retrieve a page. Thus, the measure for the response is approximated the number of nodes (pages) that will be retrieved by the range query.

The next lemma forms the basis for the analysis:

Lemma 1 *If the node n_i of the R-tree has an MBR of $n_{i,x} \times n_{i,y}$, then the probability $DA(n_{i,x}, n_{i,y})$ that this node will contribute one disk access to a point query is*

$$DA(n_{i,x}, n_{i,y}) = \text{Prob(a point query retrieves node } n_i) = n_{i,x} * n_{i,y} \tag{1}$$

Proof: Since we assume that the (point) queries are uniformly distributed in the address space and the address space is the unit square. The probability that a random point fall in the rectangle $(n_{i,x}, n_{i,y})$ is the area of the rectangle $n_{i,x} \times n_{i,y}$. ■

$DA()$ is the expected number of disk accessed that the specific node will contribute in an arbitrary point query. Notice that the *level* of the node in the R-tree is *immaterial.*

The next two lemmas calculate the expected number of disk accesses for point and rectangular queries respectively.

Lemma 2 (Point query) *For a point query, the expected number of disk accesses* $P(0,0)$ *is given by*

$$P(0,0) \ = \sum_{i=1}^{N} n_{i,x} * n_{i,y} \tag{2}$$

Proof: Every node n_i in the R-tree is represented in the native space by its minimum bounding rectangle (MBR) of size say $n_{i,x}$, $n_{i,y}$ in the x, y direction respectively. Given Lemma 1, each node of the R-tree contributes $DA()$ disk access; to calculate the average number of disk accesses that *all* the nodes of the R-tree will result into, we have to sum Eq. 1 over all the nodes. ■

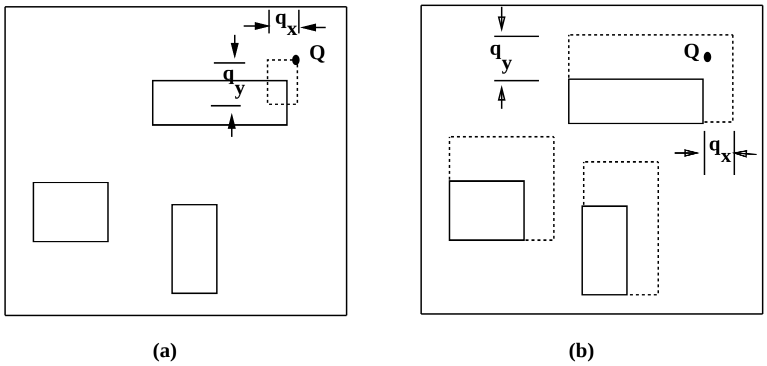

Figure 8: a) *original nodes along with rectangular query* $q_x \times q_y$ *b) Extended nodes with point query* Q

Lemma 3 (Rectangular query) *For a rectangular query* $q_x \times q_y$*, the expected number of disk accesses* $P(q_x, q_y)$ *is given by*

$$P(q_x, q_y) \ = \sum_{i=1}^{N} n_{i,x} * n_{i,y} + q_y \times \sum_{i=1}^{N} n_{i,x} + q_x * \sum_{i=1}^{N} n_{i,y} + N * q_x * q_y \tag{3}$$

Proof: A rectangular query of size $q_x \times q_y$ is equivalent to a point query, if we 'inflate' the nodes of the R-tree by q_x and q_y in the x- and y-directions respectively. Thus, the node n_i with size $n_{i,x} \times n_{i,y}$ behaves like a node of size $(n_{i,x} + q_x) \times (n_{i,y} + q_y)$. Figure 8 illustrates the idea: Figure 8(a) shows a range query $q_x \times q_y$ with upper-left corner at Q; this query is equivalent to a point query anchored at Q, as long as the data rectangles are 'inflated' as shown by the dotted lines in Figure 8(b). Applying (Eq. 2) on Figure 8(b) we obtain:

$$P(q_x, q_y) = \sum_{i=1}^{N} (n_{i,x} + q_x) * (n_{i,y} + q_y) \tag{4}$$

which gives (Eq. 3), after trivial mathematical manipulations. ■

Notice that Lemma 3 gives

$$P(q_x, q_y) = TotalArea + q_x * L_y + q_y * L_x + N * q_x * q_y \tag{5}$$

where $TotalArea = P(0,0)$ is the sum of all the areas of the nodes of the tree, and L_x, L_y are respectively the sums of x and y extents of all nodes in the R-tree.

There are several comments and observations with respect to the above formulas:

- The formula is *independent* of the details of the R-tree creation/insertion/split algorithms; it holds for packed R-trees, for R^*-trees e.t.c.
- Notice that Eq. 3 for range queries reduces to Eq. 2 for point queries, if $q_x = q_y = 0$, as expected.

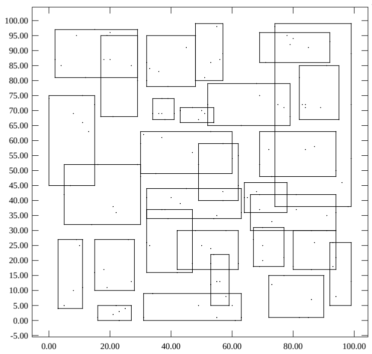

Figure 9: *MBR of nodes generated by 2D-c (2-d Hilbert through centers), for 200 random points*

- The last equation illustrates the importance of minimizing the perimeter of the R-tree nodes, in addition to the area. The larger the queries, the more important the perimeter becomes. This explains why the *lowx* packed R-tree performs well for point queries ($q_x = q_y = 0$), but not so well for larger queries. The nodes produced by the *lowx* packed R-tree (figure 4) have small area but large perimeters. Figure 9 shows the leaf nodes produced by the '2D-c' hilbert packing method, for the set of points of figure 3. Notice that the resulting nodes have smaller perimeters.

- Eq. 3 has theoretical as well as practical value: From a practical point of view, it can assist with the cost estimation and query optimization for spatial queries [1]: Maintaining only a few numbers about the R-tree (total area, total perimeter), a query optimizer can make a good estimate for the cost of a range query. Moreover, researchers working on R-trees can use Eq 3 to avoid issuing queries in their simulation studies. This eliminates one randomness factor (the query), leaving the generation and insertion order of the data as random variables. From a theoretical point of view, it makes one step towards the theoretical analysis of the R-trees; the only missing step is a good estimate of the average size and perimeter of nodes, for a given data set and a given packing/insertion/split algorithm.

5 Experimental results

To assess the merit of our proposed hilbert-based packing methods, we ran simulation experiments on a two-dimensional native space. The code for the packed R-tree is written in C under UNIX and the simulation experiments ran on a SUN SPARC station. 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 lines and/or rectangles

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

Query area $Q_{area} = q_x \times q_y$: ranged from 0 - 0.25 of the space size

Page size p: varied in the range 1Kb - 4Kb

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.

The rectangles for the synthetic datasets were generated as follows: Their centers were uniformly distributed in the unit square; their x and y sizes were uniformly distributed in the range [0,max], where max was chosen to achieve the desired data density. Data or query rectangles that were not completely inside the unit square were clipped.

In addition to synthetic data, we also used real data from the TIGER system of the U.S. Bureau of Census. One data set consisted 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. Another dataset consists of 53145 line segments, representing the roads of Long Beach, California. The data density of the MBR that covers these line segments is $d = 0.15$. We refer to this dataset as the '*LBeach*' dataset. An important observation is that the data in the TIGER dataset follow a highly *skewed* distribution.

In the following subsections we present experiments (a) comparing the response time of the best of our methods (2D-c) with the response time of older R-tree variants (dynamic or static) (b) comparing our hilbert-based packing schemes against each other, to pinpoint the best.

5.1 Comparison of 2D-c hilbert packed R-tree vs. older R-tree variants

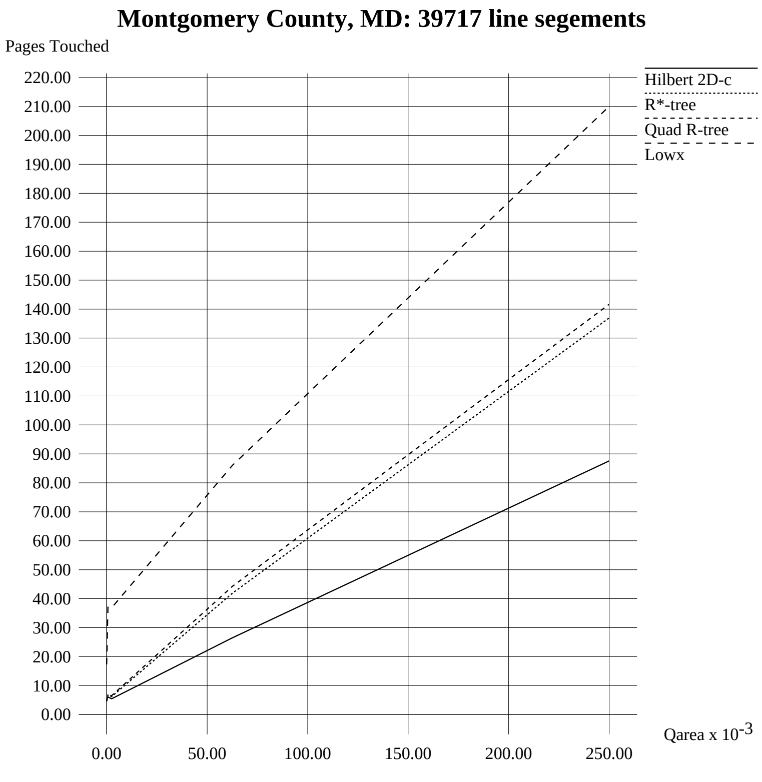

Figure 10: *Montgomery county, Maryland - real data*

Here we illustrate that the 2D-c packing method gives better response times than older R-tree variants. The fact that *lowx* packed R-tree performs worse than Dynamic designs (e.g. $R^* - tree$) mandated on us to compare our new packing methods with both Static and Dynamic designs, namely *lowx* packed R-tree, Guttman R-tree and $R^* - tree$. In our plots we omit the results

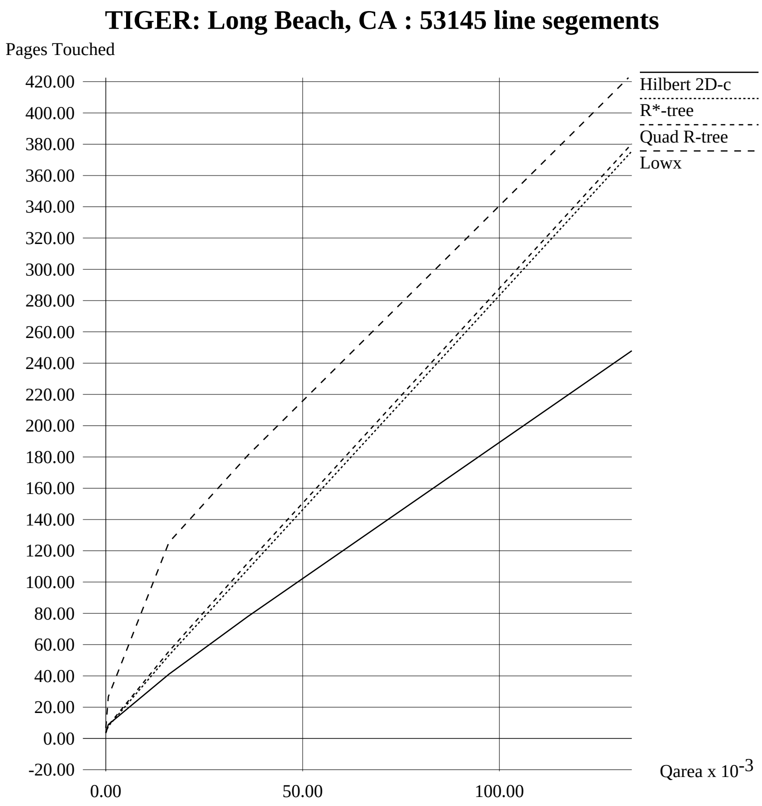

Figure 11: *Long Beach, California - real data*

of the linear-split R-tree, because the quadratic-split R-tree consistently outperformed it. The exponential-split R-tree was very slow in building the tree, and it was not used.

To avoid cluttering the plots, we only plot the best of our proposed algorithms, namely the one using the '2D-c' heuristic. The detailed results for the other, hilbert-based packing algorithms are presented in the next subsection.

We performed experiments with several data sets using different parameters. For brevity we present experiments with four sets of data. The first two data sets are from the 'TIGER' system, while the last two are 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.). Figure 10, 11 show the results for the TIGER data sets , they represent the roads of Montgomery county of Maryland and the roads of Long Beach of California respectively. Figure 12 shows the results for a synthetic data set, which contains a mix of points and rectangles; specifically 50,000 points and 10,000 rectangles; the data density is $d = 0.029$ and the page size $p = 1$Kb. The forth set (Figure 13) contains no points; only 100,000 rectangles; the data density $d = 1$ and the page size $p = 1$Kb.

All Figures 10- 13 plot the number of pages retrieved by a range query (from Eq. 3) as a

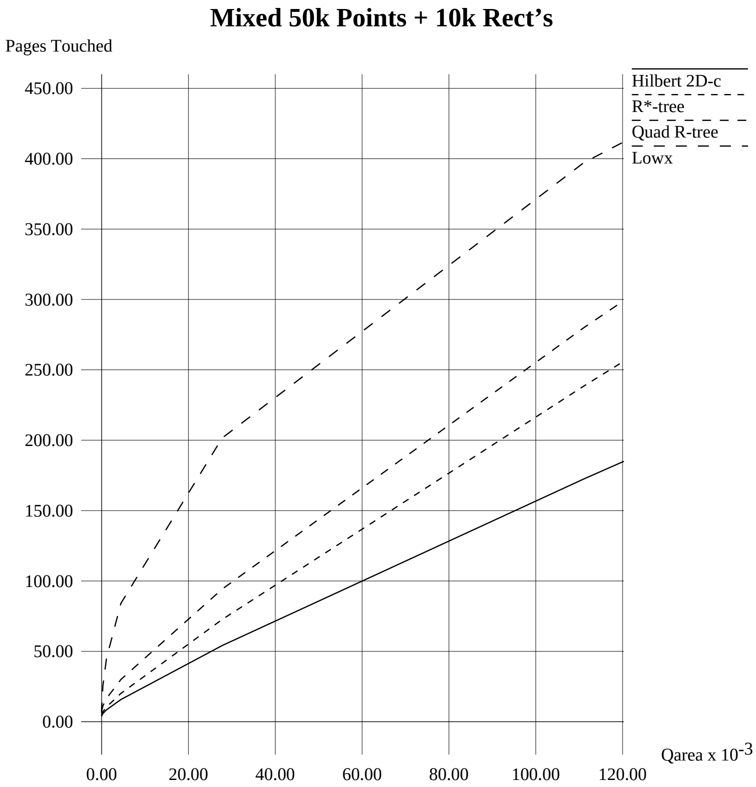

Figure 12: *Page accesses per query vs. Query area - Rectangles + Points*

function of the area of the query. A common observation is that, for point queries, all methods performs almost the same, with small differences. However, for slightly larger queries, the proposed 2D-c hilbert packed R-tree is the clear winner. The performance gap increases with the area of the queries.

The second important observation is that the performance gap seems to increase with the *skeweness* of the data distribution: for the TIGER data sets, the proposed method achieves up to 36% improvement over the next best method (R^*-tree) , and up to 58% improvement over the *lowx* packed R-tree.

Since the performance of the Static *lowx* packed R-tree (100% space utilization) is worse than the performance of the Dynamic designs (e.g. quadratic split R-tree and $R^* - tree$) we ascribe the good performance of our proposed methods not only to the higher space utilization but also to the superior clustering property of the Hilbert curve.

5.2 Comparison of hilbert-based packing schemes

Here we compare all the packing heuristics that we have introduced in this paper, namely

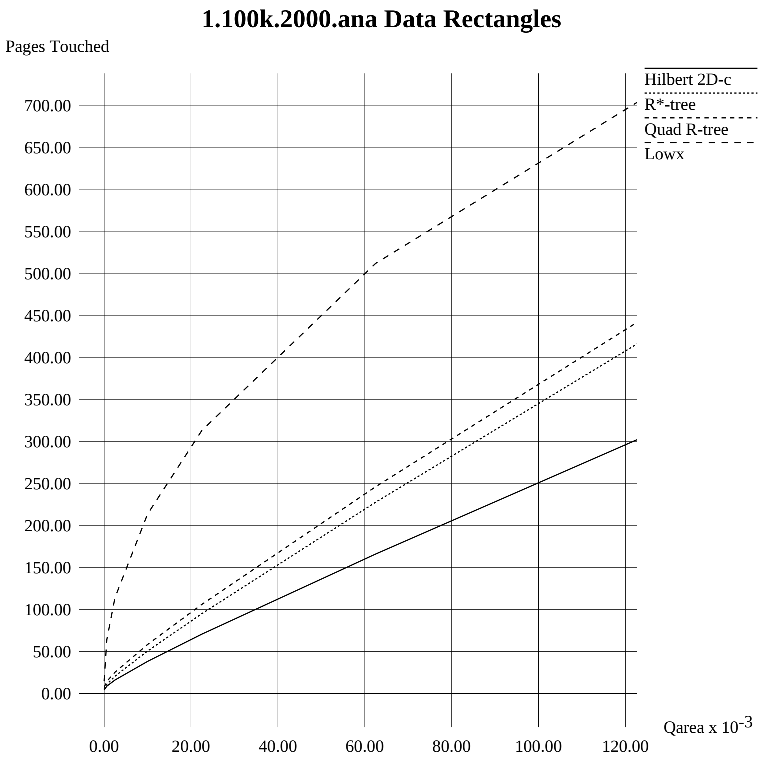

Figure 13: *Page accesses per query vs. Query area - Rectangles Only*

2D-c , 4D-xy, 4D-cd and the only heuristic that uses the z-ordering, 2Dz-c. Table 1 contains a list of these methods, along with a brief description.

Table 3 gives the response time versus the query area for all of these heuristics that use *Hilbert* order. The (synthetic) data file consists of 50K points and 10K rectangles. The page size p = 1Kb. The differences between the alternative methods are small. However, from Table 3 we see that (2D-c) does better, especially for large queries. The next best method is the (4D-cd), which uses a 4-d hilbert curve on the parameter space (center-x, center-y, diameter-x, diameter-y). The

query area Q_{area}	Hilbert 2D-c	Hilbert 4-cd	Hilbert 4D-xy
0.000000	3.74	5.10	7.04
0.000278	5.60	7.28	9.26
0.001111	8.22	10.24	12.04
0.004444	15.20	17.84	20.32
0.111111	169.76	177.06	180.54

Table 3: Comparison Between Different Schemes that uses Hilbert order

query area Q_{area}	Hilbert 2D-c	Z-order 2Dz-c
0.000000	3.74	5.98
0.000278	5.60	8.64
0.001111	8.22	11.48
0.004444	15.20	20.28
0.111111	169.76	183.56

Table 4: Schema that uses Hilbert order vs. the one uses z-order

last contender is the 4D-xy.

For the same setting, table 4 compares 2D-c that sorts the data on the *2d hilbert-value* of the centers of the data rectangles and 2Dz-c that sorts on the *2d z-value* of the center. Table 4 shows that 2D-c that uses hilbert order always performs better than 2Dz-c that uses z-order.

The relative ranking of the methods was the same for every dataset we tried; we omit the results for brevity.

6 Conclusions

We have examined algorithms to generate packed R-trees for static databases. Our algorithms try to exploit the superior clustering properties of the Hilbert curve. We propose several schemes to sort the data rectangles, before grouping them into R-tree nodes. We performed experiments with all these methods and the most promising competitors; the major conclusion is that the proposed algorithms result in better R-trees. Specifically, the most successful variation (2D-c = 2-d hilbert curve through centers) consistently outperforms the best dynamic methods, namely, the R^*-trees and the quadratic-split R-trees, as well as the only previously known static method (*lowx* packed R-tree). More importantly, the performance gap seems to be wider for real, skewed data distributions.

An additional, smaller contribution, is the derivation of (Eq. 3), From a practical point of view, it can help a query optimizer [14] give a good estimate for the cost of an R-tree index. Moreover, it makes the simulation analysis of R-trees easier and more reliable, eliminating the need to ask queries.

Future research could examine methods to achieve provably optimal packing, as well as the corresponding lower bound on the response time.

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

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

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

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

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

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

[14] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible query processing in starburst. *Proc. ACM-SIGMOD 1989 Int'l Conf. Management of Data*, pages 377–388, May 1989.

[15] 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.

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

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

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

[19] 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.

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

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

[22] 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.

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

[24] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed r-trees. *Proc. ACM SIGMOD*, May 1985.

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

[26] 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.

[27] 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.