Four-Dimensional Hilbert Curves for R-Trees

Herman Haverkort* Freek van Walderveen†

Abstract

Two-dimensional R-trees are a class of spatial index structures in which objects are arranged to enable fast window queries: report all objects that intersect a given query window. One of the most successful methods of arranging the objects in the index structure is based on sorting the objects according to the positions of their centres along a two-dimensional Hilbert space-filling curve. Alternatively one may use the coordinates of the objects' bounding boxes to represent each object by a four-dimensional point, and sort these points along a four-dimensional Hilbert-type curve. In experiments by Kamel and Faloutsos and by Arge et al. the first solution consistently outperformed the latter when applied to point data, while the latter solution clearly outperformed the first on certain artificial rectangle data. These authors did not specify which four-dimensional Hilbert-type curve was used; many exist.

In this paper we show that the results of the previous papers can be explained by the choice of the four-dimensional Hilbert-type curve that was used and by the way it was rotated in four-dimensional space. By selecting a curve that has certain properties and choosing the right rotation one can combine the strengths of the two-dimensional and the four-dimensional approach into one, while avoiding their apparent weaknesses. The effectiveness of our approach is demonstrated with experiments on various data sets. For real data taken from VLSI design, our new curve yields R-trees with query times that are better than those of R-trees that were obtained with previously used curves.

1 Introduction

In many applications one needs a spatial database to store and query objects in a plane. For example in geographic information systems, objects on the surface of the earth are stored, and one needs to be able to retrieve the objects that lie within a certain query range. A database with the components of a VLSI design constitutes another example. When such databases are big, disk access becomes a major issue. Therefore a lot of research has been done into spatial databases that are designed to be stored on disk.

R-trees, originally introduced by Guttman [5], form a class of spatial index structures that are particularly interesting as a general-purpose method to organise a variety of spatial objects. The R-tree can be understood as a multi-dimensional variant of a B-tree that stores minimum bounding boxes instead of one-dimensional keys. An R-tree is structured as follows. The minimum axis-parallel bounding boxes of the data objects are organised into leaves (pages) that can accommodate a certain number of bounding boxes. Let B be the number of bounding boxes that fit in a leaf. The leaves are then organised into a height-balanced tree, of which each node has roughly B children (the root node may have fewer children). Each node of the tree occupies a page on disk and stores, for each of its children ν, the address (page number) of ν and the bounding box of the objects stored in the subtree rooted at ν.

To find the objects intersecting a query window Q, we start at the root and recursively query all subtrees whose bounding boxes intersect Q. Whenever the query reaches a leaf, we check the bounding boxes stored in it, and for each bounding box that intersects Q we check the corresponding object. Fig. 1 shows an example of an R-tree, highlighting the nodes that are accessed to answer an example query. R-trees can also be used to find the objects that are nearest to a query range Q; the process is similar to searching the bounding boxes in a range around Q that is grown until the nearest objects are found.

R-trees have been proven quite effective in practice, but the query efficiency depends on how exactly the rectangles are ordered in the tree. In practice it is typically possible to keep all internal nodes of the tree cached in memory, while the leaves often have to be retrieved from disk. Therefore the number of leaves accessed is the most important factor determining the query efficiency of an R-tree. Intuitively, one would like to make sure that the rectangles that are stored in any particular leaf ν lie close to each other in the plane. Thus the bounding box of ν will be small, thus the chance that any particular query intersects that

*Dept. of Mathematics and Computer Science, Eindhoven University of Technology, The Netherlands. cs.herman@haverkort.net

†Dept. of Mathematics and Computer Science, Eindhoven University of Technology, The Netherlands. freek@vanwal.nl

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

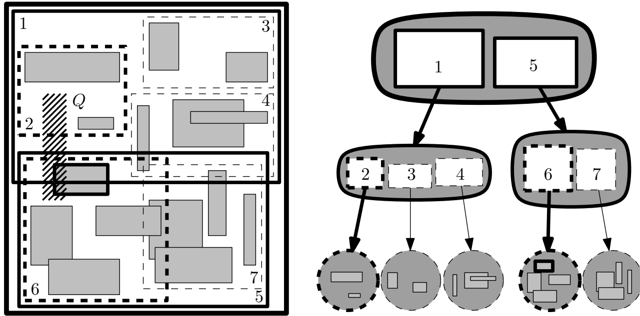


Figure 1: An example of an R-tree. When queried with the hatched query range Q, the nodes marked by bold outlines will be visited.

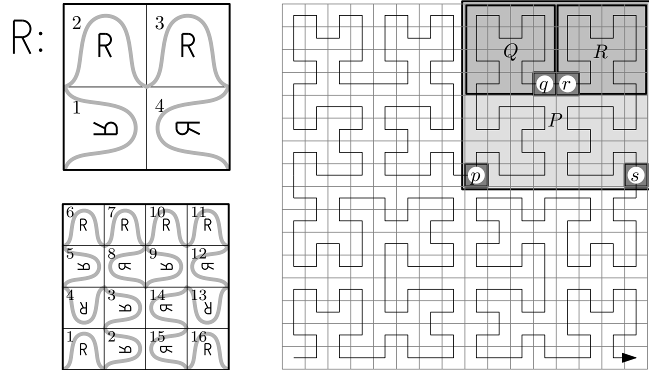


Figure 2: Top left: the definition of the Hilbert-order. The grey curve illustrates the scanning order of the quadrants. Bottom left: two levels of subdivision. Right: four levels of subdivision. The first cell p and the last cell s of quadrant P are in the corners. The last cell q of Q and the first cell r of R share an edge.

bounding box is small, and thus the chance that ν needs to be retrieved from disk is small. The need to optimise the organisation of rectangles in an R-tree has spawned a lot of research: many algorithms have been designed to construct R-trees and to insert or delete rectangles in them. For an overview see Manolopoulos et al. [10].

A particularly easy and successful method is based on sorting the set of input rectangles into a linear order. Then the tree can be constructed by packing the input rectangles into leaves in the order in which they have been sorted, and simply putting a balanced tree of degree roughly B on top of them. This results in an R-tree construction algorithm that is as fast as sorting and scanning the input once. Furthermore, it is easy to maintain the structure by keeping the rectangles in the tree in order as in a B-tree. The main consideration with this approach is how to define the linear order, and several heuristics and variations on this approach have been explored [4, 8, 9, 13]. In particular, the Hilbert-order used by Kamel and Faloutsos [8] was shown to result in efficient queries in experiments.

1.1 A Hilbert R-tree on a set of points. We define a *scanning order* $\prec$ of points in the unit square as follows. We define a set of rules, each of which specifies (i) a scanning order $\prec$ of the four quadrants of a square, and (ii) for each quadrant, which rule is to be applied to establish the scanning order within that quadrant. We choose a starting rule R and apply it to the unit square. Fig. 2 illustrates the starting rule defining the Hilbert-order. It puts the quadrants of the unit square in the order lower left, upper left, upper right, lower right. To the upper left and the upper right quadrants, rule R is applied recursively. To the lower quadrants, we apply a rotated and mirrored version of rule R.

By applying these rules recursively, we can order the squares of an arbitrarily fine grid. Drawing a curve through the centres of these squares gives an arbitrarily fine approximation of Hilbert's space-filling curve [7], see Fig. 2. For two points a and b, we say $a \prec b$ (a precedes b) if and only if, through recursive subdivision of the unit square according to these rules, we can find two squares A and B such that $a \in A$, $b \in B$, and $A \prec B$. When a point lies on a vertical boundary between quadrants, we assume it belongs to the quadrant to the right; when a point lies on a horizontal boundary between quadrants, we assume it belongs to the quadrant above.

1.2 A Hilbert R-tree on a set of rectangles. Unfortunately the approach described above only works for point data. To build an R-tree on a set of objects in the plane that are not points, we can map the bounding boxes of the objects to points and put the objects in the tree according to the Hilbert-order on those points. Kamel and Faloutsos investigated three ways to do this:

- map bounding boxes $[x_{\text{mn}}, x_{\text{mx}}] \times [y_{\text{mn}}, y_{\text{mx}}]$ to their centre points $\frac{1}{2}(x_{\text{mn}} + x_{\text{mx}}, y_{\text{mn}} + y_{\text{mx}})$ and use the Hilbert-order defined above (the *two-dimensional mapping*, or *H2* for short);
- map bounding boxes $[x_{\text{mn}}, x_{\text{mx}}] \times [y_{\text{mn}}, y_{\text{mx}}]$ to four-dimensional points $(x_{\text{mn}}, y_{\text{mn}}, x_{\text{mx}}, y_{\text{mx}})$ and order them according to a Hilbert-like scanning order of the four-dimensional unit hypercube (the *four-dimensional xy-mapping*, *H4xy* for short);
- map bounding boxes $[x_{\text{mn}}, x_{\text{mx}}] \times [y_{\text{mn}}, y_{\text{mx}}]$ to four-dimensional points (c_x, c_y, d_x, d_y), were $c_x = \frac{1}{2}(x_{\text{mn}} + x_{\text{mx}})$, $c_y = \frac{1}{2}(y_{\text{mn}} + y_{\text{mx}})$, $d_x = x_{\text{mx}} -$

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

x_{mn}, and $d_y = y_{\text{mx}} - y_{\text{mn}}$; order these points according to a Hilbert-like scanning order of the four-dimensional unit hypercube (the *four-dimensional cd-mapping*, *H4cd* for short).

The two-dimensional approach ignores the widths and heights of the input objects' bounding boxes. As a result, a vertical line segment and a horizontal line segment with the same midpoint would always be put together in one leaf, creating a leaf with a large bounding box, see Fig. 3. The four-dimensional approach may avoid this problem by grouping boxes together that are similar in both location *and* orientation. However, this raises the question of how exactly to define a scanning order of the four-dimensional unit hypercube. The two-dimensional Hilbert-order can be generalised to higher dimensions in many different, equally plausible ways, and there is no agreement in the literature of what would constitute *the* four-dimensional Hilbert-order. Different implementations of multi-dimensional Hilbert-orders may sort points differently.

Kamel and Faloutsos [8] and Arge et al. [2] show comparisons between the two-dimensional approach H2 and one or both of the four-dimensional approaches, but they do not discuss what four-dimensional Hilbert-like scanning order they used. Kamel and Faloutsos concluded from their experiments that H2 works best in practice. The results of Arge et al. are consistent with that conclusion, except for extreme, artificially constructed data sets, which may be unlike anything one would expect to find in practice. On those extreme data sets, the H2 approach resulted in very bad query times while the H4xy approach did well.

Neither Kamel and Faloutsos nor Arge et al. showed any advantages of a four-dimensional approach applied to a *practical* data set. On the contrary, their results seem to show that the four-dimensional approach is clearly *worse* in practice. This is quite counterintuitive. Even if the four-dimensional approach does not do any good on 'nice' data (which consists mostly of very small objects), it is hard to see why taking height and width information into account when ordering small bounding boxes would have to do any harm.

1.3 Our results. Considering the counterintuitive results of Kamel and Faloutsos and Arge et al. with unspecified four-dimensional scanning orders, we decided to investigate the effect of the choice of the four-dimensional scanning order on the query efficiency of the resulting R-trees. We found that the choice of scanning order has a significant impact. In fact the impact is such that Kamel and Faloutsos could have arrived at entirely different conclusions if they had tried different four-dimensional Hilbert-like scanning orders.

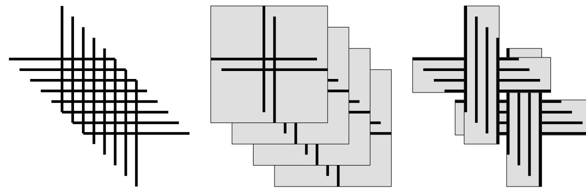


Figure 3: Left: a set of objects (line segments). Centre: when objects with approximately the same centre are grouped together into leaves, regardless of their orientation, leaves with large bounding boxes result. Right: when the orientation of the objects is taken into account when packing them together, smaller bounding boxes are possible.

In particular, we found a four-dimensional Hilbert-like scanning order that seems to combine the strengths of the previously used scanning orders while avoiding their apparent weaknesses. On relatively 'nice' data, the new order results in a query efficiency that is as good as with the two-dimensional approach, or only slightly worse. On extreme data the new order results in query times that are as good as with the previously used four-dimensional scanning orders. Moreover, it turns out that there are data sets in the practice of VLSI design for which the new order gives considerably better results than the two-dimensional approach.

2 Defining a four-dimensional scanning order

To define a four-dimensional Hilbert-like scanning order we use the approach of Alber and Niedermeier [1]. They define a class of multi-dimensional scanning orders that maintain the most characteristic properties of the two-dimensional Hilbert-order. To explain their method, we first discuss the relevant properties of the two-dimensional Hilbert-order. Next we illustrate how one can define multi-dimensional scanning orders with an example in three dimensions. Then we give definitions of four-dimensional scanning orders, and explain the algorithms we used to find new scanning orders.

2.1 Two dimensions. Recall the two-dimensional Hilbert-order described in Section 1.1: it is defined by ordering the quadrants of the unit square, and by specifying, for each quadrant, how to mirror and/or rotate the order within that quadrant. The Hilbert-order has two special properties, illustrated in Fig. 2.

Property 2.1. *Consider a grid of square cells that results from an arbitrarily deep recursive application of the rules that generate the scanning order, and consider any square P to which the rules have been applied, on any level of the recursion. The cells p and s in P that*

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

come first and last in the scanning order, are in two corners of P.

PROPERTY 2.2. *Let Q and R be two quadrants of a square P as defined above, such that R is the immediate successor of Q in the scanning order. Then the first cell r of R shares an edge with the last cell q of Q.*

The latter is a useful property when the scanning order is used to make R-trees: it guarantees that consecutive cells in the Hilbert-order are adjacent in space. Thus points that end up in the same leaf of the R-tree are likely to be close to each other in space, and thus they have a small bounding box.

2.2 Three dimensions. Alber and Niedermeier generalise the Hilbert order to higher dimensions. Their d-dimensional scanning orders are based on recursively ordering the 2^d 'hyperquadrants' of the d-dimensional unit hypercube, while maintaining the above properties 2.1 and 2.2. They find that there are 1536 structurally different three-dimensional scanning orders that have these properties. To describe a particular order, one needs to specify in which order the 'hyperquadrants' of the unit hypercube appear, and how the order is mirrored and/or rotated within each hyperquadrant. Alber and Niedermeier use a permutation-based notation for this purpose, which we will illustrate with an example in three dimensions.

Consider a scanning order that puts the octants of the three-dimensional unit cube in the order LTF (left top front), LTH (left top hind), RTH (right top hind), RTF, RBF (right bottom front), RBH, LBH, LBF, see Fig. 4. Using 0 to encode left, bottom, and front, and 1 to encode right, top, and hind, we can also present this order as 010-011-111-110-100-101-001-000. Note that in this case, the first and the last octant are adjacent.

Now we need to specify how to order the suboctants inside the left-top-front octant A_1. The first of these will also be the first suboctant of the unit hypercube as a whole; therefore it must be in a corner (to ensure Property 2.1), so it must be the suboctant A_{11} on the left-top-front side of A_1, see Fig. 4. The last suboctant of A_1 must be adjacent to the next octant A_2 (to ensure Property 2.2) and to A_{11}, so it must be suboctant A_{18} on the left-top-hind side of A_1. This leaves two possible orders of suboctants within A_1 that can be obtained by mirroring and/or rotating the order of the octants of the unit cube: it must be either 010-110-100-000-001-101-111-011 (as in Fig. 4, top), or 010-000-100-110-111-101-001-011 (as in Fig. 4, bottom).

To specify which order we take, we describe how to permute the order of the octants of the unit cube (010-011-111-110-100-101-001-000) to get the order of the suboctants inside A_1. Such a permutation is given as a set of sequences of indices, where a sequence of k indices $(i_0 i_1 ... i_{k-1})$, with $i_j \in \{1, 2, ..., 2^d\}$, indicates that octant i_j in the permuted order is octant $i_{(j+1) \bmod k}$ in the original order, for $0 \leqslant j < k$. Thus the two choices of permutations for the first octant of the unit cube are (2 4 8) (3 5 7) and (2 8) (3 5).

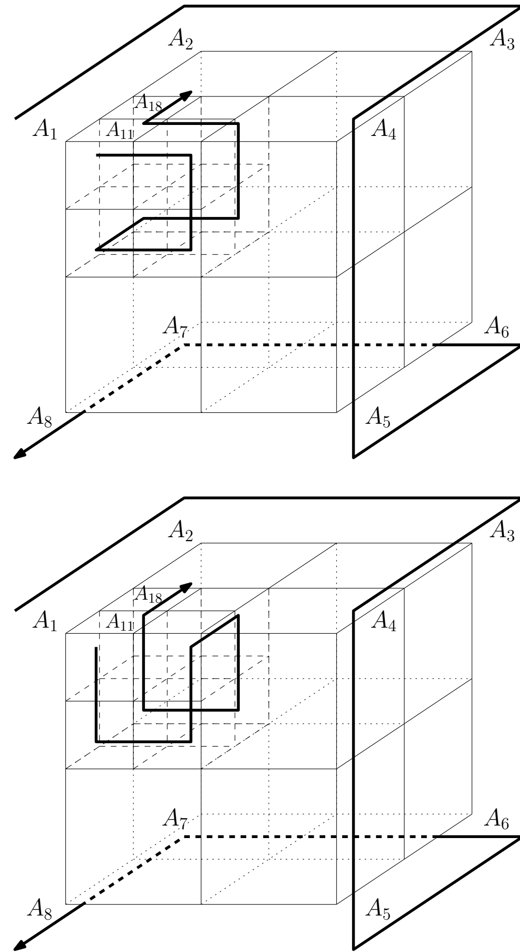
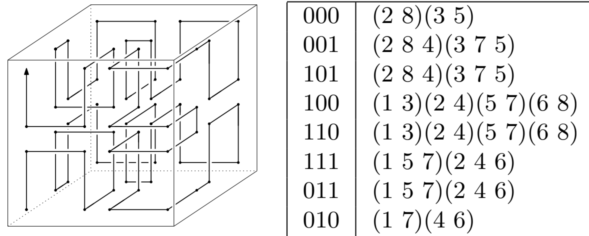


Figure 4: Example of a top-level order, showing different orders for the first octant.

To specify a full scanning order, we specify the top-level order and we specify a permutation of that order for every octant. See Fig. 5 for an example.

2.3 Four dimensions (H4cd). In the H4cd approach we distinguish four dimensions, which correspond to the following attributes of input boxes $[x_{\mathrm{mn}}, x_{\mathrm{mx}}] \times [y_{\mathrm{mn}}, y_{\mathrm{mx}}]$: (1) horizontal location $\frac{1}{2}(x_{\mathrm{mn}} + x_{\mathrm{mx}})$ (from left to right); (2) vertical location $\frac{1}{2}(y_{\mathrm{mn}} + y_{\mathrm{mx}})$ (from bottom to top); (3) width $(x_{\mathrm{mx}} - x_{\mathrm{mn}})$ (from narrow to wide); (4) height $(y_{\mathrm{mx}} - y_{\mathrm{mn}})$ (from short to tall). We identify a hyperquadrant H of

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.



000	(2 8)(3 5)
001	(2 8 4)(3 7 5)
101	(2 8 4)(3 7 5)
100	(1 3)(2 4)(5 7)(6 8)
110	(1 3)(2 4)(5 7)(6 8)
111	(1 5 7)(2 4 6)
011	(1 5 7)(2 4 6)
010	(1 7)(4 6)

Figure 5: Example of a fully specified scanning order in three dimensions. The scanning order is defined by the table, with the left column defining the top-level order and the right column defining the permutations within the octants.

the four-dimensional unit hypercube by a four-digit binary number $b_1b_2b_3b_4$, where $b_i = 0$ if H is on the small (left, bottom, narrow, short) side of dimension i, and $b_i = 1$ if H is on the large (right, top, wide, tall) side of dimension i. Fig. 6 shows the definitions of a four-dimensional scanning order by Alber and Niedermeier and the order of Moore's software [11] (H4cdM).

When we order bounding boxes of point data (with width and height dimension zero) with the known four-dimensional scanning orders, then their order is quite different from their order according to the two-dimensional approach. For example, Fig. 7 shows the order in which sixteen points of the type (x, y) appear when we order the corresponding points $(x, y, 0, 0)$ with Alber and Niedermeier's scanning order. As it turns out, the scanning order of such points in a grid of 16 squares contains 'jumps': sometimes cells that are consecutive in the scanning order are not adjacent in the plane. As a result, two points that lie very close to each other in the scanning order—and are therefore likely to be packed into the same leaf—can be very far apart in the plane, causing the leaf to have a large bounding box. We believe this may explain why previous authors found that the four-dimensional approach did not work well in practice. Therefore we decided to look for four-dimensional orders with the following property:

PROPERTY 2.3. *For any two points $p_1 = (x_1, y_1, 0, 0)$ and $p_2 = (x_2, y_2, 0, 0)$ we have $p_1 \prec p_2$ in the four-dimensional scanning order if and only if $(x_1, y_1) \prec (x_2, y_2)$ according to the two-dimensional Hilbert-order.*

To find a scanning order that has properties 2.1, 2.2 *and* 2.3, we implemented an algorithm to search the space of four-dimensional scanning orders within the framework of Alber and Niedermeier. Our search method is explained in the next section, and resulted in the order shown in Fig. 6 (bottom).

We also experimented with a rotated version of the H4cdAN-order (Fig. 6). We call this version H4cdANR:

Alber-Niedermeier order (H4cdAN)	
0000	(2 16)(3 13)(6 12)(7 9)
0010	(3 15)(4 16)(5 9)(6 10)
0110	(3 15)(4 16)(5 9)(6 10)
0100	(1 3 13 11 9 7)(2 4 14 12 10 8)(5 15)(6 16)
1100	(1 3 13 11 9 7)(2 4 14 12 10 8)(5 15)(6 16)
1110	(1 5 13 9)(2 6 14 10)(3 11 15 7)(4 12 16 8)
1010	(1 5 13 9)(2 6 14 10)(3 11 15 7)(4 12 16 8)
1000	(1 7)(4 6)(10 16)(11 13)
1001	(1 7)(4 6)(10 16)(11 13)
1011	(1 9 13 5)(2 10 14 6)(3 7 15 11)(4 8 16 12)
1111	(1 9 13 5)(2 10 14 6)(3 7 15 11)(4 8 16 12)
1101	(1 11)(2 12)(3 5 7 9 15 13)(4 6 8 10 16 14)
0101	(1 11)(2 12)(3 5 7 9 15 13)(4 6 8 10 16 14)
0111	(1 13)(2 14)(7 11)(8 12)
0011	(1 13)(2 14)(7 11)(8 12)
0001	(1 15)(4 14)(5 11)(8 10)

Moore order (H4cdM)	
0000	(2 4 8 16)(3 5 9 15)(6 12 10 14)(7 13)
1000	(2 8)(3 9)(4 16)(5 15)(6 10)(12 14)
1100	(2 8)(3 9)(4 16)(5 15)(6 10)(12 14)
0100	(1 3 13 5)(2 14 12 8)(6 16)(7 15 11 9)
0110	(1 3 13 5)(2 14 12 8)(6 16)(7 15 11 9)
1110	(1 5 11 15)(2 4 12 10)(3 13 9 7)(6 14 16 8)
1010	(1 5 11 15)(2 4 12 10)(3 13 9 7)(6 14 16 8)
0010	(1 7)(2 8)(3 5)(4 6)(9 15)(10 16)(11 13)(12 14)
0011	(1 7)(2 8)(3 5)(4 6)(9 15)(10 16)(11 13)(12 14)
1011	(1 9 11 3)(2 16 12 6)(4 8 10 14)(5 7 15 13)
1111	(1 9 11 3)(2 16 12 6)(4 8 10 14)(5 7 15 13)
0111	(1 11)(2 6 8 10)(3 5 9 15)(4 12 16 14)
0101	(1 11)(2 6 8 10)(3 5 9 15)(4 12 16 14)
1101	(1 13)(2 12)(3 5)(7 11)(8 14)(9 15)
1001	(1 13)(2 12)(3 5)(7 11)(8 14)(9 15)
0001	(1 15 13 9)(2 14 12 8)(3 11 5 7)(4 10)

New order (H4cdNew)	
0000	(2 16)(3 9)(4 8)(6 12)(7 13)(10 14)
0010	(3 15)(4 16)(5 9)(6 10)
0110	(2 8)(3 9)(4 16)(5 15)(6 10)(12 14)
0100	(1 3)(5 13)(6 16)(7 15)(8 14)(9 11)
1100	(1 3 15 11 9 5)(2 14 10 12 8 4)(6 16)(7 13)
1110	(1 5 11 15)(2 4 12 10)(3 13 9 7)(6 14 16 8)
1010	(1 5 3 11 15 9)(2 12)(4 6 14 10 16 8)(7 13)
1000	(1 7)(4 6)(10 16)(11 13)
1001	(1 7)(4 6)(10 16)(11 13)
1011	(1 9 13 11 3 7)(2 8 16 12 14 6)(4 10)(5 15)
1111	(1 9 11 3)(2 16 12 6)(4 8 10 14)(5 7 15 13)
1101	(1 11)(2 6 8 12 16 14)(3 7 5 9 13 15)(4 10)
0101	(1 11)(2 10)(3 9)(4 12)(6 8)(14 16)
0111	(1 13)(2 12)(3 5)(7 11)(8 14)(9 15)
0011	(1 13)(2 14)(7 11)(8 12)
0001	(1 15)(3 7)(4 10)(5 11)(8 14)(9 13)

Figure 6: Definitions of three four-dimensional scanning orders.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

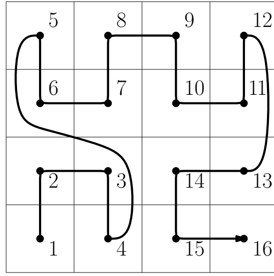


Figure 7: The order of point data according to the H4cd approach with the scanning order of Alber and Niedermeier.

it has top-level order 0000-1000-1100-0100-0110-1110-1010-0010-0011-1011-1111-0111-0101-1101-1001-0001; the permutations are the same as in the H4cdAN-order.

2.4 Four dimensions (H4xy). For the H4xy approach, we experimented with the orders by Alber and Niedermeier (H4xyAN) and by Moore (H4xyM), and with a new scanning order (H4xyNew, different from H4cdNew) that satisfies properties 2.1 and 2.2 and the H4xy-analogue of Property 2.3:

PROPERTY 2.3. *(H4xy-version) For any two points $p_1 = (x_1, y_1, x_1, y_1)$ and $p_2 = (x_2, y_2, x_2, y_2)$ we have $p_1 \prec p_2$ in the four-dimensional scanning order if and only if $(x_1, y_1) \prec (x_2, y_2)$ according to the two-dimensional Hilbert-order.*

Because the H4xy approach performed worse than the H4cd approach in all of our experiments, we omit a precise definition of H4xyNew.

3 Finding good scanning orders

We will now explain how we searched for scanning orders that satisfy properties 2.1, 2.2 and 2.3.

3.1 Testing Property 2.3. To test if a four-dimensional scanning order satisfies Property 2.3, we proceed as follows. We run an algorithm that handles requests to verify that a two-dimensional and a four-dimensional scanning order *match*, that is:

- *Condition 1:* the four quadrants 00, 01, 11, and 10 appear in the two-dimensional order in the same order as the corresponding hyperquadrants 0000, 0100, 1100, and 1000 in the four-dimensional order;
- *Condition 2:* in each of these quadrants, the permuted two-dimensional order matches the permuted four-dimensional order in the corresponding hyperquadrant.

Our algorithm maintains a queue of such requests. As long as there are requests in the queue, the algorithm extracts one from the queue, verifies Condition 1 for this request, and generates requests for the permuted orders in each quadrant to verify Condition 2; these requests are then appended to the queue. Each request is specified by the permutations of the top-level orders in the two-dimensional quadrant and the four-dimensional hyperquadrant that should be verified to match.

Clearly a four-dimensional scanning order satisfies Property 2.3 if and only if it *matches* the two-dimensional Hilbert order, which has top-level order 00-01-11-10 and permutations (2 4), $\emptyset$, $\emptyset$, (1 3). Therefore the algorithm starts with a queue of one request $(\mathcal{I}, \mathcal{I})$, which is to verify that the two-dimensional scanning order permuted by the identity permutation $\mathcal{I}$ matches the four-dimensional scanning order permuted by $\mathcal{I}$. As the algorithm is running, more requests are generated. As soon as the algorithm fails to verify Condition 1 while handling a request, we can conclude that the order being tested does not satisfy Property 2.3.

However, if the order satisfies Property 2.3, the algorithm as just described would never finish, because it adds four verification requests to the queue for every request it extracts. Therefore we modify the algorithm slightly: when a new request is generated, we only insert it into the queue if it was never inserted before. The algorithm maintains a list of requests previously inserted to check this condition. Since the total number of different possible requests is finite (at most 3072 are possible[1]), the queue eventually runs empty and the algorithm terminates. For pseudocode see Fig. 8.

We will now prove that when the algorithm terminates without failing on any verification request, we can conclude that the order being tested satisfies Property 2.3. Let $p'_1 = (x_1, y_1)$ and $p'_2 = (x_2, y_2)$ be two points in the unit square, specified by binary numbers $(0.x_{11}x_{12}x_{13}..., 0.y_{11}y_{12}y_{13}...)$ and $(0.x_{21}x_{22}x_{23}..., 0.y_{21}y_{22}y_{23}...)$. Let p_1 and p_2 be the corresponding four-dimensional points $(x_1, y_1, 0, 0)$ and $(x_2, y_2, 0, 0)$. Let k be the smallest i for which $x_{1i} \neq x_{2i}$ or $y_{1i} \neq y_{2i}$. Define $a_0 = b_0 = \mathcal{I}$, and define for $i \in \{1, 2, 3, ..., k-1\}$:

$$a_i = a_{i-1} \circ \alpha_{(A \circ a_{i-1})^{-1}(x_{1i}y_{1i})}$$
$$b_i = b_{i-1} \circ \beta_{(B \circ b_{i-1})^{-1}(x_{1i}y_{1i}00)}.$$

(See the code for the definition of A, α, B, and β.) Note that a_i specifies the transformation of the two-di-

[1] The number of possible combinations of rotations and reflections of a d-dimensional Hilbert-like scanning order is $2^d d!$; therefore there can be at most $2^2 2! \cdot 2^4 4! = 3072$ different requests. With a more refined analysis and algorithm the number of different requests can be reduced to 384.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

```
MatchOrders(A, α, B, β)
    ▷ A : {1,...,4} → {0,1}^2 specifies the 2D top-level order;
    ▷ α_i : {1,...,4} → {1,...,4} is its transformation (permutation) within quadrant A(i);
    ▷ B : {1,...,16} → {0,1}^4 specifies the 4D top-level order;
    ▷ β_i : {1,...,16} → {1,...,16} is its transformation (permutation) within hyperquadrant B(i);
    ▷ I = identity transformation
 1  Initialise queue Q with one element (I, I)
 2  Initialise list L with one element (I, I)
 3  while Q is not empty
 4       do Extract request (a, b) from Q     ▷ to match 2D order permuted by a and 4D order permuted by b
 5          pj ← 0                           ▷ record rank of last-visited hyperquadrant in 4D
 6          for i ← 1 to 4                   ▷ go through four quadrants in 2D
 7               do Let q1q2 be the two digits of (A ∘ a)(i)
 8                  j ← (B ∘ b)^{-1}(q1q2 00)  ▷ locate corresponding hyperquadrant in 4D
 9                  if j > pj                 ▷ verify Condition 1: j must increase when i increases.
10                     then                   ▷ generate request to verify Condition 2
11                          if (a ∘ α_i, b ∘ β_j) ∉ L
12                             then insert (a ∘ α_i, b ∘ β_j) in Q and L
13                          pj ← j
14                     else return false
15  return true
```

Figure 8: The algorithm to verify that a four-dimensional scanning order satisfies Property 2.3.

mensional scanning order in the quadrant i levels down in the recursion from the unit square, that contains both p'_1 and p'_2. Similarly, b_i specifies the transformations of the four-dimensional scanning order in the hyperquadrant that contains both p_1 and p_2. The reader may now verify that the algorithm must have generated verification requests $(a_1, b_1), (a_2, b_2), ..., (a_{k-1}, b_{k-1})$. When the request (a_{k-1}, b_{k-1}) was extracted from the queue, the algorithm verified Condition 1, which implies that $(A \circ a_{k-1})^{-1}(x_{1k}y_{1k}) < (A \circ a_{k-1})^{-1}(x_{2k}y_{2k})$ if and only if $(B \circ b_{k-1})^{-1}(x_{1k}y_{1k}00) < (B \circ b_{k-1})^{-1}(x_{2k}y_{2k}00)$. Thus $p_1 \prec p_2$ in the four-dimensional scanning order if and only if $p'_1 \prec p'_2$ in the two-dimensional order.

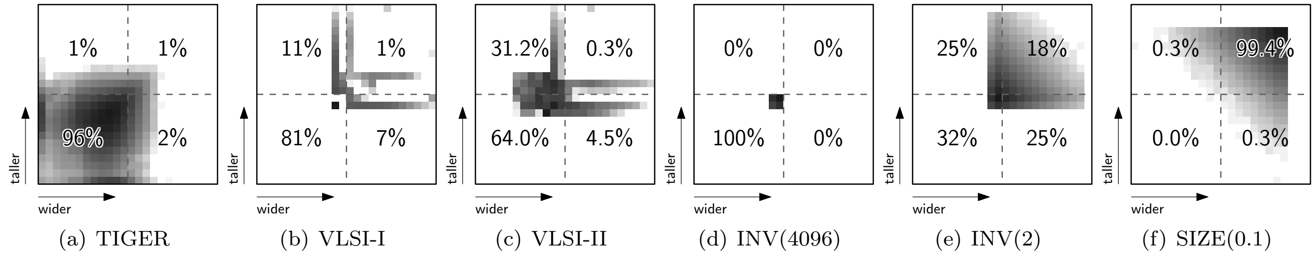
Therefore, if the algorithm terminates successfully on a given four-dimensional order, it has Property 2.3.

3.2 Generating scanning orders. A scanning order can be partially defined by specifying: (i) the top-level order; (ii) for each hyperquadrant Q, the subquadrant where the order enters Q and the subquadrant where the order leaves Q; (iii) for each of the hyperquadrants 0000, 0100, 1000, and 1100, the transformation (permutation) of the order inside that hyperquadrant. Note that the only part that is not specified in such a partial definition are the transformations inside the remaining twelve hyperquadrants. Note also that such a partial definition is sufficient to determine if a scanning order satisfies Condition 1 on the top level and the second level of recursion. The specification of the subquadrants of entry and exit in each quadrant serves to set up a framework within which transformations inside hyperquadrants can be chosen independently from each other while maintaining Property 2.2.

For our experiments we generated all partially defined scanning orders that start in quadrant 0000 and end in quadrant 1000, and satisfy Condition 1 on the top level and the second level of recursion. For each of these partially defined scanning orders, we searched the possible choices for the remaining twelve unspecified transformations to get a fully specified scanning order that also satisfies Condition 1 further down in the recursion. Here we tried a heuristic approach. We start from an initial choice for the twelve transformations, and then repeatedly test the scanning order with the algorithm of Fig. 8 and change the transformation that causes the matching algorithm to fail—ensuring termination by never changing the transformation of a quadrant to one that was tried before. Although this heuristic approach is not exhaustive (it may fail to find a solution if one exists), it resulted in 218 fully specified scanning orders based on 218 different partially defined scanning orders. We observed that scanning orders that are correct on the seventh level of recursion sometimes fail to satisfy Property 2.3 on the eighth level, while all scanning orders that were found correct on the eighth level, were in fact correct on any level.

We then ran some small experiments similar to those described in the next section to determine which

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.



(a) TIGER (b) VLSI-I (c) VLSI-II (d) INV(4096) (e) INV(2) (f) SIZE(0.1)

Figure 9: The distribution of the heights and widths of rectangles in the various data sets in our experiments, and in an artificial SIZE(0.1) data set like those used by Arge et al. Each plot consists of 24 × 24 cells. The cell in the x-th column from the left and the y-th row from below represents rectangles with width between $2^{x-25}w$ and $2^{x-24}w$ and height between $2^{y-25}w$ and $2^{y-24}w$, where w is the width of the bounding box of the complete data set. The darker a cell, the more rectangles of the corresponding width and height range are found in the data set. Rectangles with width less than $2^{-24}w$ count towards the leftmost cells and rectangles with height less than $2^{-24}w$ count towards the bottommost cells. The dashed lines form the boundary between rectangles smaller than $2^{-12}w$ and rectangles larger than $2^{-12}w$. For the TIGER data this means that the boundary between small and large rectangles lies at approximately 300 metres.

scanning order, out of the 218 scanning orders found, seemed to yield the R-trees with the best query performance. The order shown in Fig. 6 (bottom) appeared to be one of the most promising and was then tested more extensively, as described below.

4 Experimental results

We tested the scanning orders defined above by building R-trees on them following the approach by Kamel and Faloutsos: we sorted the input rectangles according to the scanning order and then packed the rectangles into leaves of B rectangles in order. This is not necessarily the best way of using scanning orders to build R-trees: sometimes it may be better to use the scanning order only to determine in which order the input rectangles are fed to a more elaborate packing algorithm (see for example DeWitt et al. [4] or Van den Bercken and Seeger [3]). Nevertheless we stick with the approach by Kamel and Faloutsos so that we compare the effectiveness of different scanning orders by themselves, rather than their combinations with various selections of other heuristics.

We used five types of data sets for our experiments:

INV(a) 10 million random rectangles, uniformly distributed in the unit square, with each rectangle's height and width independently set to $1/rnd$, where rnd is a random real number uniformly distributed between a and 9486. This results in a data set with many small rectangles and some larger rectangles. The parameter a is used to control the size of the larger rectangles with respect to a fixed query size.

KF a data set of 50 000 points and 10 000 rectangles of size at most 0.02 × 0.02, uniformly distributed in the unit square (as in Kamel and Faloutsos [8]).

TIGER 16.7 million bounding boxes of road segments of five eastern states of the USA (as in Arge et al. [2]).

VLSI-I 6.0 million rectangles of a VLSI design.

VLSI-II 35.8 million rectangles of a VLSI design.

The size distribution of the rectangles in these data sets is illustrated in Figure 9. In the TIGER data, almost all rectangles are small. In the VLSI data sets, most rectangles are small, some are big. The artificial INV data sets share these properties: the parameter a allows us to adjust the variation in rectangle size while maintaining that large rectangles are less frequent than small rectangles. Our data sets seem to be more realistic in this sense than an artificial SIZE(0.1) data set as in Arge et al. [2], which contains as many big rectangles as small ones.

We did a number of experiments in which we constructed R-trees on the data sets listed above using the various scanning orders. To be able to compare our results with Arge et al. [2], we built all R-trees with a block size (number of bounding boxes per node) of 113 rectangles, except the R-trees on the KF data, where we used a block size of 50 rectangles because Kamel and Faloutsos [8] experimented with a smaller block size than Arge et al.[2] On each R-tree we performed

[2]We also did experiments where we used different block sizes to see if the block size affects the ranking of the different R-tree structures in terms of query efficiency. We found that larger block sizes tend to make the differences between the R-trees somewhat more pronounced, but their ranking does not change.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

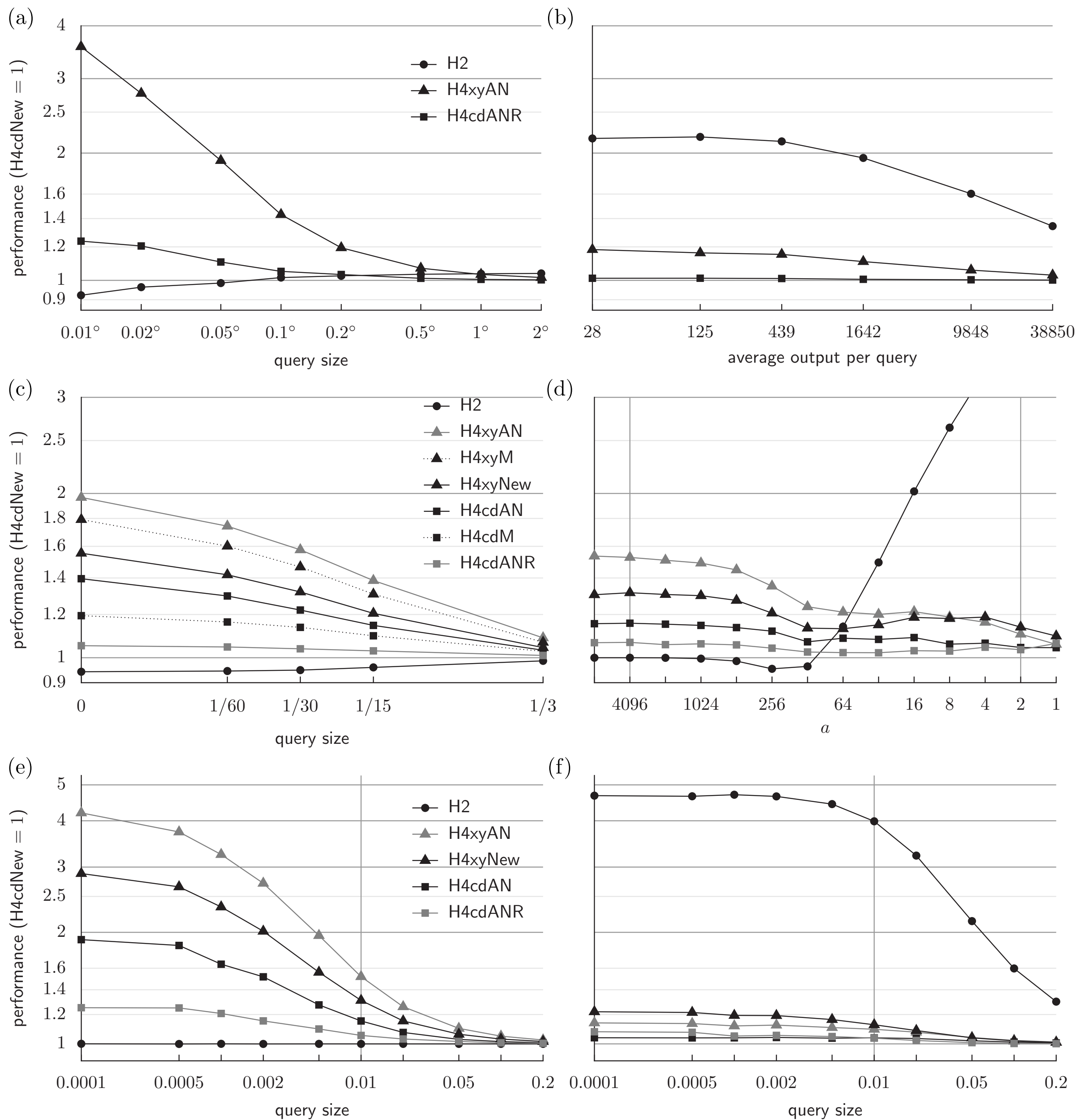


Figure 10: Number of leaves accessed for square queries of various widths on (a) TIGER data; (b) VLSI-I; (c) KF; (d) queries of size 0.01×0.01 on INV(a) data for various values of a; queries of various widths on (e) INV(4096); (f) INV(2). All results shown relative to those with H4cdNew = 1.

an experiment with a fixed query size or several experiments with several query sizes. In each such experiment we performed 1000 uniformly distributed square window queries and measured the query efficiency in two ways: by counting the number of leaves accessed (considering that internal nodes may be cached in main memory and only leaves are fetched from disk [2]), and by counting the *total* number of nodes accessed (leaves and internal nodes alike). We found that including or excluding internal nodes from the count has hardly any influence on

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

the relative performance of the R-trees, so we present results for only one way of counting, namely leaves only.

The results are shown in Fig. 10. For each query experiment the figure shows the number of leaves read from various R-trees relative to H4cdNew, the structure based on our new scanning order. Thus the results for H4cdNew are always 1 by definition and therefore they are omitted from the figures.

The results on the TIGER data (Fig. 10(a)) can be compared to those of Arge et al. [2]. The experiments presented there correspond roughly to our experiments with query sizes of $1°$ to $2°$ (1 to 2 degrees longitude and latitude). While for the larger queries, all structures perform well (as observed by Arge et al.), performance differs considerably for smaller queries, with H4xyAN being particularly ineffective.

On VLSI data (Fig. 10(b) and Table 1), our solutions H4cdANR and H4cdNew clearly outperform the others.

The experiments on the KF data (Fig. 10(c)) illustrate the importance of choosing the right scanning order, and the right mapping of bounding box attributes to four-dimensional coordinates. Our results on H2, H4cdAN and H4xyAN are similar to those presented by Kamel and Faloutsos [8], which led them to conclude that H2 works best in practice. We also tried the H4xy and the H4cd approach with the scanning order produced by Moore's implementation (H4cdM and H4xyM), with similar results. However, the performance of H4cdAN and our variant with two coordinates swapped (H4cdANR), may differ by more than 30%. Our solutions H4cdANR and H4cdNew almost match the performance of the two-dimensional approach (H2) on the KF data.

With our tests on various INV(a)-data sets we investigated the influence of two variables: the width and height of the square query ranges, and the parameter a that controls the size distribution of the rectangles in the data. Fig. 10(d) shows the results with a fixed query width (0.01) for various values of a. For $a = 4096$ and for $a = 2$ we ran tests with various query sizes, see Figures 10(e) and (f). Note that the relative ordering of the results on INV(4096)-data is similar to those on TIGER data, and the results on INV(2)-data are similar to the results on VLSI-I data. This suggests that the INV data sets indeed cover a range of data sets that are representative for data sets encountered in practice.

We observe the following trends throughout the experiments. Out of the R-trees based on a four-dimensional scanning order, H4cdNew is always best. This leaves H2 and H4cdNew as our main competitors. On sparse, uniform data (INV(a) for higher values of a, TIGER, KF), H2 and H4cdNew both perform very well.

R-tree	leaves read
H2	1.37
H4xyAN	1.18
H4xyNew	1.16
H4cdANR	1.00

Table 1: Efficiency of queries of size 0.005×0.005 on VLSI-II, scaled to lie within the unit square, with 1026 answers per query on average. (H4cdNew = 1)

On dense, non-uniform data (INV(a) for lower values of a, VLSI) H4cdNew performs well or very well, while H2 is very poor. In general the ranking of the various R-trees does not change with the query size, but with smaller query sizes the differences are more pronounced.

5 Conclusions, directions for further research

There is no reason to be satisfied with Hilbert-R-trees based on mapping bounding boxes to two-dimensional points, or based on an arbitrary four-dimensional scanning order. By choosing the mapping of two-dimensional rectangle coordinates to four-dimensional point coordinates wisely and using our new four-dimensional scanning order, one gets better results.

Nevertheless the question what is *the* best four-dimensional scanning order for R-tree construction remains unanswered. The space of possible (and practical) four-dimensional scanning orders is huge. Because assessing the quality of an order by experiments is rather time-consuming, we could only search a small part of that space in a rather rough way. We only generated a subset of possible partially specified scanning orders (it may be possible to start and end in other hyper-quadrants), and for each of them we only generated at most one fully specified scanning order, using a heuristic that is not guaranteed to find a solution if one exists. Our experiments revealed unanswered theoretical questions about the nature of the search space: is it true that every scanning order that has properties 2.1, 2.2 and 2.3 on the eighth level of recursion, must have these properties on any level of recursion? More importantly, not only within the framework of Alber and Niedermeier can we find many scanning orders, there are many more. For example Peano's curve [12] can be generalised to four dimensions in a straightforward way. Preliminary experiments with such a scanning order suggest that it performs quite well, but not as well as our H4cdNew scanning order. Several other scanning orders have been defined in two dimensions (see our paper on two-dimensional orders for an overview [6]), and they, too, may possibly be generalised to four dimensions.

To guide the search it would be helpful to have qual-

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

ity measures for four-dimensional scanning orders that can be analysed and computed theoretically, similar to those which we defined for two-dimensional scanning orders [6]. This would not only be useful to speed up the search for good scanning orders, it could also ensure that the "best" scanning order is selected on the basis of more objective criteria than good average performance on a small test set. Note that a mere generalisation of two-dimensional measures to four dimensions would not be good enough for our purposes: we need a quality measure for *four-dimensional* scanning orders that are used to make bounding boxes in *two dimensions*.

References

[1] J. Alber, R. Niedermeier: On Multidimensional Curves with Hilbert Property. *Theory of Computing Systems* 33:295–312 (2000).

[2] L. Arge, M. de Berg, H. J. Haverkort, and K. Yi. The Priority R-tree: a practically efficient and worst-case optimal R-tree. In *Proc. ACM SIG Management of Data (SIGMOD)*, pages 347–358, 2004.

[3] J. van den Bercken, B. Seeger. An Evaluation of Generic Bulk Loading Techniques. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, pages 461–470, 2001.

[4] D. J. DeWitt, N. Kabra, J. Luo, J. M. Patel, and J.-B. Yu. Client-server paradise. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, pages 558–569, 1994.

[5] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM Special Interest Group on Management of Data (SIGMOD)*, pages 47–57, 1984.

[6] H. Haverkort and F. van Walderveen. Locality and bounding-box quality of two-dimensional space-filling curves. In *Proc. 16th Eur. Symp. on Algorithms (ESA)*, 2008. Full manuscript at arXiv:0806.4787 [cs.CG].

[7] D. Hilbert. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Math. Ann.* 38 (1891), 459–460.

[8] I. Kamel and C. Faloutsos. On packing R-trees. In *Proc. Int. Conf. on Information and Knowledge Management (CIKM)*, pages 490–499, 1993.

[9] S. T. Leutenegger, M. A. López, and J. Edgington. STR: A simple and efficient algorithm for R-tree packing. In *Proc. IEEE Int. Conf. on Data Engineering (ICDE)*, pages 497–506, 1996.

[10] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, Y. Theodoridis. *R-trees: Theory and Applications.* Springer, 2005.

[11] D. Moore. *Fast Hilbert Curve Generation, Sorting, and Range Queries.* `http://web.archive.org/web/www.caam.rice.edu/~dougm/twiddle/Hilbert/`

[12] G. Peano. Sur une courbe, qui remplit toute une aire plane. *Math. Ann.* 36 (1890), 157–160.

[13] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. In *Proc. ACM Special Interest Group on Management of Data (SIGMOD)*, pages 17–31, 1985.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.