The Priority R-Tree: A Practically Efficient and Worst-Case-Optimal R-Tree

Lars Arge

Mark de Berg

Herman J. Haverkort

Ke Yi

institute of information and computing sciences, utrecht university

technical report UU-CS-2004-022

www.cs.uu.nl

The Priority R-Tree: A Practically Efficient and Worst-Case Optimal R-Tree

Lars Arge* Mark de Berg† Herman J. Haverkort‡ Ke Yi§

April 19, 2004

Abstract

We present the Priority R-tree, or PR-tree, which is the first R-tree variant that always answers a window query using $O((N/B)^{1-1/d} + T/B)$ I/Os, where N is the number of d-dimensional (hyper-) rectangles stored in the R-tree, B is the disk block size, and T is the output size. This is provably asymptotically optimal and significantly better than other R-tree variants, where a query may visit all N/B leaves in the tree even when $T = 0$. We also present an extensive experimental study of the practical performance of the PR-tree using both real-life and synthetic data. This study shows that the PR-tree performs similar to the best known R-tree variants on real-life and relatively nicely distributed data, but outperforms them significantly on more extreme data.

1 Introduction

Spatial data naturally arise in numerous applications, including geographical information systems, computer-aided design, computer vision and robotics. Therefore spatial database systems designed to store, manage, and manipulate spatial data have received considerable attention over the years. Since these databases often involve massive datasets, disk based index structures for spatial data have been researched extensively—see e.g. the survey by Gaede and Günther [11].

Especially the R-tree [13] and its numerous variants (see e.g. the recent survey by Manolopoulos et al. [19]) have emerged as practically efficient indexing methods. In this paper we present the Priority R-tree, or *PR-tree*, which is the first R-tree variant that is not only practically efficient but also provably asymptotically optimal.

*Lars Arge, Department of Computer Science, Duke University, Box 90129, Durham, NC 27708-0129, USA, large@cs.duke.edu. Supported in part by the National Science Foundation through RI grant EIA–9972879, CAREER grant CCR–9984099, ITR grant EIA–0112849, and U.S.–Germany Cooperative Research Program grant INT–0129182.

†Mark de Berg, Department of Computer Science, TU Eindhoven, P.O.Box 513, 5600 MB Eindhoven, The Netherlands, m.t.d.berg@tue.nl.

‡Herman J. Haverkort, Institute of Information and Computing Sciences, Utrecht University, PO Box 80 089, 3508 TB Utrecht, The Netherlands, herman@cs.uu.nl. Supported by the Netherlands' Organization for Scientific Research (NWO).

§Ke Yi, Department of Computer Science, Duke University, Box 90129, Durham, NC 27708-0129, USA, yike@cs.duke.edu.

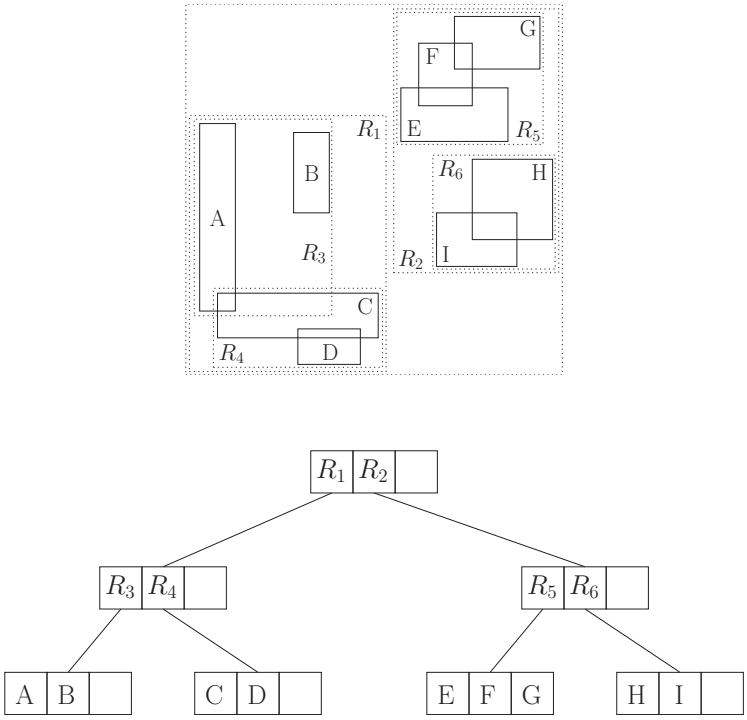

Figure 1: R-tree constructed on rectangles A, B, C, ..., I (block size = 3).

1.1 Background and previous results

Since objects stored in a spatial database can be rather complex they are often approximated by simpler objects, and spatial indexes are then built on these approximations. The most commonly used approximation is the minimal bounding box: the smallest axis-parallel (hyper-) rectangle that contains the object. The R-tree, originally proposed by Guttman [13], is an index for such rectangles. It is a height-balanced multi-way tree similar to a B-tree [5, 9], where each node (except for the root) has degree $\Theta(B)$. Each leaf contains $\Theta(B)$ data rectangles (each possibly with a pointer to the original data) and all leaves are on the same level of the tree; each internal node v contains pointers to its $\Theta(B)$ children, as well as for each child a minimal bounding box covering all rectangles in the leaves of the subtree rooted in that child. Figure 1 shows an example. If B is the number of rectangles that fits in a disk block, an R-tree on N rectangles occupies $\Theta(N/B)$ disk blocks and has height $\Theta(\log_B N)$. Many types of queries can be answered efficiently using an R-tree, including the common query called a window query: Given a query rectangle Q, retrieve all rectangles that intersect Q. To answer such a query we simply start at the root of the R-tree and recursively visit all nodes with minimal bounding boxes intersecting Q; when encountering a leaf l we report all data rectangles in l intersecting Q.

Guttman gave several algorithms for updating an R-tree in $O(\log_B N)$ I/Os using B-tree-like algorithms [13]. Since there is no unique R-tree for a given dataset, and because the window query performance intuitively depends on the amount of overlap between minimal bounding boxes in the nodes of the tree, it is natural to try to minimize bounding box overlap during updates. This has led to the development of many heuristic update algorithms; see for

example [6, 16, 23] or refer to the surveys in [11, 19]. Several specialized algorithms for bulk-loading an R-tree have also been developed [7, 10, 12, 15, 18, 22]. Most of these algorithms use $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os (the number of I/Os needed to sort N elements), where M is the number of rectangles that fits in main memory, which is much less than the $O(N\log_B N)$ I/Os needed to build the index by repeated insertion. Furthermore, they typically produce R-trees with better space utilization and query performance than R-trees built using repeated insertion. For example, while experimental results have shown that the average space utilization of dynamically maintained R-trees is between 50% and 70% [6], most bulk-loading algorithms are capable of obtaining over 95% space utilization. After bulk-loading an R-tree it can of course be updated using the standard R-tree updating algorithms. However, in that case its query efficiency and space utilization may degenerate over time.

One common class of R-tree bulk-loading algorithms work by sorting the rectangles according to some global one-dimensional criterion, placing them in the leaves in that order, and then building the rest of the index *bottom-up* level-by-level [10, 15, 18]. In two dimensions, the so-called packed Hilbert R-tree of Kamel and Faloutsos [15], which sorts the rectangles according to the Hilbert values of their centers, has been shown to be especially query-efficient in practice. The Hilbert value of a point p is the length of the fractal Hilbert space-filling curve from the origin to p. The Hilbert curve is very good at clustering spatially close rectangles together, leading to a good index. A variant of the packed Hilbert R-tree, which also takes the extent of the rectangles into account (rather than just the center), is the four-dimensional Hilbert R-tree [15]; in this structure each rectangle $((x_{\min}, y_{min}), (x_{\max}, y_{\max}))$ is first mapped to the four-dimensional point $(x_{\min}, y_{\min}, x_{\max}, y_{\max})$ and then the rectangles are sorted by the positions of these points on the four-dimensional Hilbert curve. Experimentally the four-dimensional Hilbert R-tree has been shown to behave slightly worse than the packed Hilbert R-tree for nicely distributed realistic data [15]. However, intuitively, it is less vulnerable to more extreme datasets because it also takes the extent of the input rectangles into account.

Algorithms that bulk-load R-trees in a *top-down* manner have also been developed. These algorithms work by recursively trying to find a good partition of the data [7, 12]. The so-called Top-down Greedy Split (TGS) algorithm of García, López and Leutenegger [12] has been shown to result in especially query-efficient R-trees (TGS R-trees). To build the root of (a subtree of) an R-tree on a given set of rectangles, this algorithm repeatedly partitions the rectangles into two sets, until they are divided into B subsets of (approximately) equal size. Each subset's bounding box is stored in the root, and subtrees are constructed recursively on each of the subsets. Each of the binary partitions takes a set of rectangles and splits it into two subsets based on one of several one-dimensional orderings; in two dimensions, the orderings considered are those by $x_{\min}, y_{\min}, x_{\max}$ and $y_{\max}$. For each such ordering, the algorithm calculates, for each of $O(B)$ possible partitioning possibilities, the sum of the areas of the bounding boxes of the two subsets that would result from the partition. Then it applies the binary partition that minimizes that sum.[1]

While the TGS R-tree has been shown to have slightly better query performance than other R-tree variants, the construction algorithm uses many more I/Os since it needs to scan all the rectangles in order to make a binary partition. In fact, in the worst case the algorithm may take $O(N\log_B N)$ I/Os. However, in practice, the fact that each partition decision is

[1] García et al. describe several variants of the top-down greedy method. They found the one described here to be the most efficient in practice [12]. In order to achieve close to 100% space utilization, the size of the subsets that are created is actually rounded up to the nearest power of B (except for one remainder set). As a result, one node on each level, including the root, may have less than B children.

binary effectively means that the algorithm uses $O(\frac{N}{B}\log_2 N)$ I/Os.

While much work has been done on evaluating the practical query performance of the R-tree variants mentioned above, very little is known about their theoretical worst-case performance. Most theoretical work on R-trees is concerned with estimating the expected cost of queries under assumptions such as uniform distribution of the input and/or the queries, or assuming that the input are points rather than rectangles. See the recent survey by Manolopoulos et al. [19]. The first bulk-loading algorithm with a non-trivial guarantee on the resulting worst-case query performance was given only recently by Agarwal et al. [2]. In d dimensions their algorithm constructs an R-tree that answers a window query in $O((N/B)^{1-1/d}+T\log_B N)$ I/Os, where T is the number of reported rectangles. However, this still leaves a gap to the $\Omega((N/B)^{1-1/d}+T/B)$ lower bound on the number of I/Os needed to answer a window query [2, 17]. If the input consists of points rather than rectangles, then worst-case optimal query performance can be achieved with e.g. a kdB-tree [21] or an O-tree [17]. Unfortunately, it seems hard to modify these structures to work for rectangles. Finally, Agarwal et al. [2], as well as Haverkort et al. [14], also developed a number of R-trees that have good worst-case query performance under certain conditions on the input.

1.2 Our results

In Section 2 we present a new R-tree variant, which we call a *Priority R-tree* or *PR-tree* for short. We call our structure the Priority R-tree because our bulk-loading algorithm utilizes so-called priority rectangles in a way similar to the recent structure by Agarwal et al. [2]. Window queries can be answered in $O((N/B)^{1-1/d}+T/B)$ I/Os on a PR-tree, and the index is thus the first R-tree variant that answers queries with an asymptotically optimal number of I/Os in the worst case. To contrast this to previous R-tree bulk-loading algorithms, we also construct a set of rectangles and a query with zero output, such that all $\Theta(N/B)$ leaves of a packed Hilbert R-tree, a four-dimensional Hilbert R-tree, or a TGS R-tree need to be visited to answer the query. We also show how to bulk-load the PR-tree efficiently, using only $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os. After bulk-loading, a PR-tree can be updated in $O(\log_B N)$ I/Os using the standard R-tree updating algorithms, but without maintaining its query efficiency. Alternatively, the external logarithmic method [4, 20] can be used to develop a structure that supports insertions and deletions in $O(\log_B \frac{N}{M}+\frac{1}{B}(\log_{M/B}\frac{N}{B})(\log_2\frac{N}{M}))$ and $O(\log_B\frac{N}{M})$ I/Os amortized, respectively, while maintaining the optimal query performance.

In Section 3 we present an extensive experimental study of the practical performance of the PR-tree using both real-life and synthetic data. We compare the performance of our index on two-dimensional rectangles to the packed Hilbert R-tree, the four-dimensional Hilbert R-tree, and the TGS R-tree. Overall, our experiments show that all these R-trees answer queries in more or less the same number of I/Os on relatively square and uniformly distributed rectangles. However, on more extreme data—large rectangles, rectangles with high aspect ratios, or non-uniformly distributed rectangles—the PR-tree (and sometimes also the four-dimensional Hilbert R-tree) outperforms the others significantly. On a special worst-case dataset the PR-tree outperforms all of them by well over an order of magnitude.

2 The Priority R-tree

In this section we describe the PR-tree. For simplicity, we first describe a two-dimensional pseudo-PR-tree in Section 2.1. The pseudo-PR-tree answers window queries efficiently but is

not a real R-tree, since it does not have all leaves on the same level. In Section 2.2 we show how to obtain a real two-dimensional PR-tree from the pseudo-PR-tree, and in Section 2.3 we discuss how to extend the PR-tree to d dimensions. In Section 2.4 we explain how a pseudo-PR-tree can serve as the basis of a structure that supports efficient insertions and deletions while maintaining optimal query efficiency. Finally, in Section 2.5 we show that a query on the packed Hilbert R-tree, the four-dimensional Hilbert R-tree, as well as the TGS R-tree can be forced to visit all leaves even if $T = 0$.

2.1 Two-dimensional pseudo-PR-trees

In this section we describe the two-dimensional pseudo-PR-tree. Like an R-tree, a pseudo-PR-tree has the input rectangles in the leaves and each internal node ν contains a minimal bounding box for each of its children ν_c. However, unlike an R-tree, not all the leaves are on the same level of the tree and internal nodes only have degree six (rather than $\Theta(B)$).

The basic idea of a pseudo-PR-tree is (similar to the four-dimensional Hilbert R-tree) to view an input rectangle $((x_{\min}, y_{\min}), (x_{\max}, y_{\max}))$ as a four-dimensional point $(x_{\min}, y_{\min}, x_{\max}, y_{\max})$. The pseudo-PR-tree is then basically just a kd-tree on the N points corresponding to the N input rectangles, except that four extra leaves are added below each internal node. Intuitively, these so-called *priority leaves* contain the extreme B points (rectangles) in each of the four dimensions. Note that the four-dimensional kd-tree can easily be mapped back to an R-tree-like structure, simply by replacing the split value in each kd-tree node ν with the minimal bounding box of the input rectangles stored in the subtree rooted in ν. The idea of using priority leaves was introduced in a recent structure by Agarwal et al. [2], they used priority leaves of size one rather than B.

In section 2.1.1 below we give a precise definition of the pseudo-PR-tree, and in section Section 2.1.2 we show that it can be used to answer a window query in $O(\sqrt{N/B} + T/B)$ I/Os. In Section 2.1.3 we describe how to construct the structure I/O-efficiently.

2.1.1 The Structure

Let $S = \{R_1, \ldots, R_N\}$ be a set of N rectangles in the plane and assume for simplicity that no two of the coordinates defining the rectangles are equal. We define $R_i^* = (x_{\min}(R_i), y_{\min}(R_i), x_{\max}(R_i), y_{\max}(R_i))$ to be the mapping of $R_i = ((x_{\min}(R_i), y_{\min}(R_i)), (x_{\max}(R_i), y_{\max}(R_i)))$ to a point in four dimensions, and define S^* to be the N points corresponding to S.

A pseudo-PR-tree $\mathcal{T}_S$ on S is defined recursively: if S contains at most B rectangles, $\mathcal{T}_S$ consists of a single leaf; otherwise, $\mathcal{T}_S$ consists of a node ν with six children, namely four priority leaves and two recursive pseudo-PR-trees. For each child ν_c, we let ν store the minimal bounding box of all input rectangles stored in the subtree rooted in ν_c. The node ν and the priority leaves below it are constructed as follows: The first priority leaf $\nu_p^{x_{\min}}$ contains the B rectangles in S with minimal $x_{\min}$-coordinates, the second $\nu_p^{y_{\min}}$ the B rectangles among the remaining rectangles with minimal $y_{\min}$-coordinates, the third $\nu_p^{x_{\max}}$ the B rectangles among the remaining rectangles with maximal $x_{\max}$-coordinates, and finally the fourth $\nu_p^{y_{\max}}$ the B rectangles among the remaining rectangles with maximal $y_{\max}$-coordinates. Thus the priority leaves contain the "extreme" rectangles in S, namely the ones with leftmost left edges, bottommost bottom edges, rightmost right edges, and topmost top edges.[2] After constructing

[2] S may not contain enough rectangles to put B rectangles in each of the four priority leaves. In that case, we may assume that we can still put at least $B/4$ in each of them, since otherwise we could just construct a

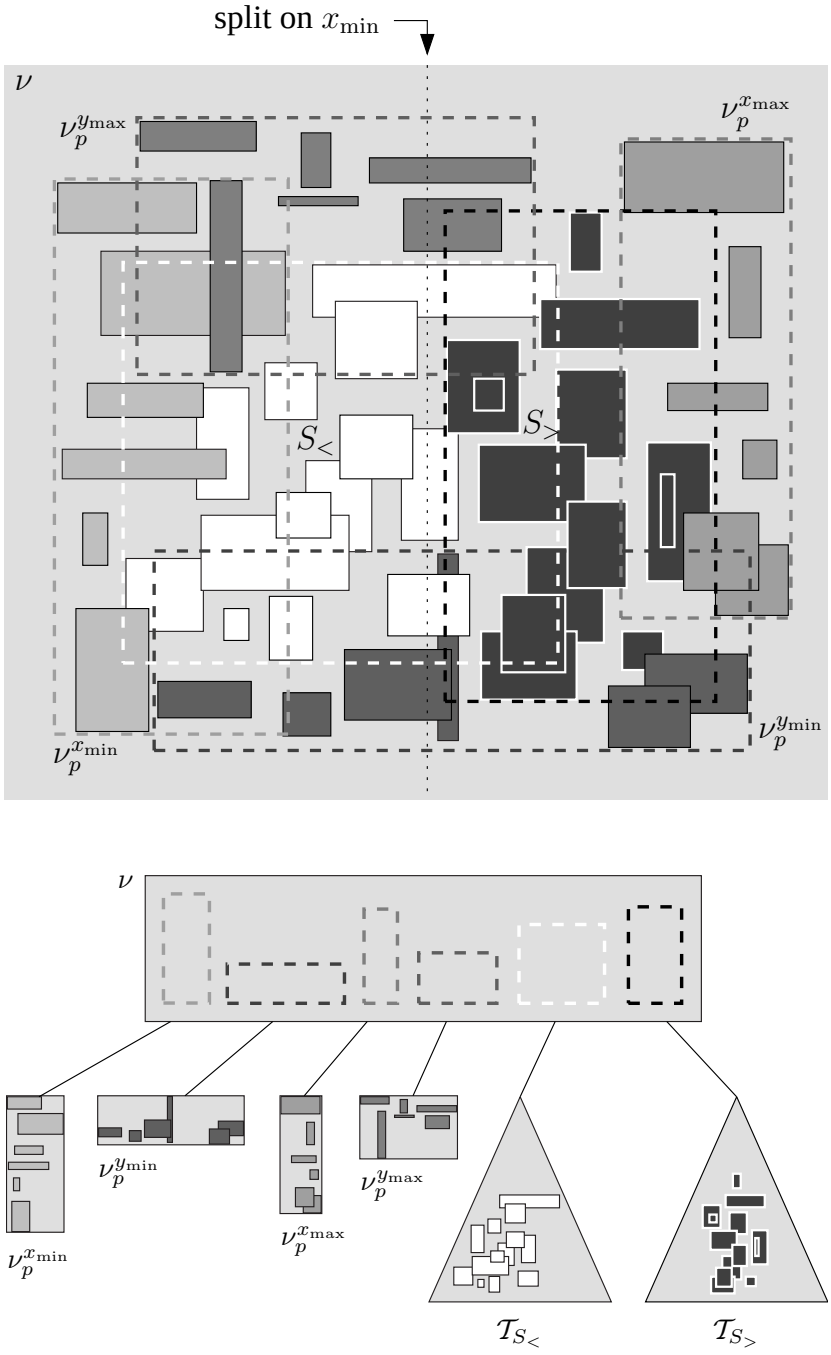

Figure 2: The construction of an internal node in a pseudo-PR-tree.

the priority leaves, we divide the set S_r of remaining rectangles (if any) into two subsets, $S_<$ and $S_>$, of approximately the same size and recursively construct pseudo-PR-trees $\mathcal{T}_{S_<}$ and $\mathcal{T}_{S_>}$. The division is performed using the $x_{\min}, y_{\min}, x_{\max}$, or $y_{\max}$-coordinate in a round-robin fashion, as if we were building a four-dimensional kd-tree on S_r^*, that is, when constructing

single leaf.

the root of $\mathcal{T}_S$ we divide based on the $x_{\min}$-values, the next level of recursion based on the $y_{\min}$-values, then based on the $x_{\max}$-values, on the $y_{\max}$-values, on the $x_{\min}$-values, and so on. Refer to Figure 2 for an example. Note that dividing according to, say, $x_{\min}$ corresponds to dividing based on a vertical line ℓ such that half of the rectangles in S_r have their left edge to the left of ℓ and half of them have their left edge to the right of ℓ.

We store each node or leaf of $\mathcal{T}_S$ in $O(1)$ disk blocks, and since at least four out of every six leaves contain $\Theta(B)$ rectangles we obtain the following (in Section 2.1.3 we discuss how to guarantee that almost every leaf is full).

Lemma 2.1 *A pseudo-PR-tree on a set of N rectangles in the plane occupies $O(N/B)$ disk blocks.*

2.1.2 Query complexity

We answer a window query Q on a pseudo-PR-tree exactly as on an R-tree by recursively visiting all nodes with minimal bounding boxes intersecting Q. However, unlike for known R-tree variants, for the pseudo-PR-tree we can prove a non-trivial (in fact, optimal) bound on the number of I/Os performed by this procedure.

Lemma 2.2 *A window query on a pseudo-PR-tree on N rectangles in the plane uses $O(\sqrt{N/B} + T/B)$ I/Os in the worst case.*

Proof: Let $\mathcal{T}_S$ be a pseudo-PR-tree on a set S of N rectangles in the plane. To prove the query bound, we bound the number of nodes in $\mathcal{T}_S$ that are "kd-nodes", i.e. not priority leaves, and are visited in order to answer a query with a rectangular range Q; the total number of leaves visited is at most a factor of four larger.

We first note that $O(T/B)$ is a bound on the number of nodes ν visited where all rectangles in at least one of the priority leaves below ν's parent are reported. Thus we just need to bound the number of visited kd-nodes where this is not the case.

Let μ be the parent of a node ν such that none of the priority leaves of μ are reported completely, that is, each priority leaf μ_p of μ contains at least one rectangle not intersecting Q. Each such rectangle E can be separated from Q by a line containing one of the sides of Q—refer to Figure 3. Assume without loss of generality that this is the vertical line $x = x_{\min}(Q)$ through the left edge of Q, that is, E's right edge lies to the left of Q's left edge, so that $x_{\max}(E) \leqslant x_{\min}(Q)$. This means that the point E^* in four-dimensional space corresponding to E lies to the left of the axis-parallel hyperplane H that intersects the $x_{\max}$-axis at $x_{\min}(Q)$. Now recall that $\mathcal{T}_S$ is basically a four-dimensional kd-tree on S^* (with priority leaves added), and thus that a four-dimensional region R^4_μ can be associated with μ. Since the query Q visits μ, there must also be at least one rectangle F in the subtree rooted at μ that has $x_{\max}(F) > x_{\min}(Q)$, so that F^* lies to the right of H. It follows that R^4_μ contains points on both sides of H and therefore H must intersect R^4_μ.

Now observe that the rectangles in the priority leaf $\mu_p^{x_{\max}}$ cannot be separated from Q by the line $x = x_{\min}(Q)$ through the left edge of Q: Rectangles in $\mu_p^{x_{\max}}$ are extreme in the positive x-direction, so if one of them lies completely to the left of Q, then all rectangles in μ's children—including ν—would lie to the left of Q; in that case ν would not be visited. Since (by definition of ν) not all rectangles in $\mu_p^{x_{\max}}$ intersect Q, there must be a line through one of Q's other sides, say the horizontal line $y = y_{\max}(Q)$, that separates Q from a rectangle G in $\mu_p^{x_{\max}}$. Hence, the hyperplane H' that cuts the $y_{\min}$-axis at $y_{\max}(Q)$ also intersects R^4_μ.

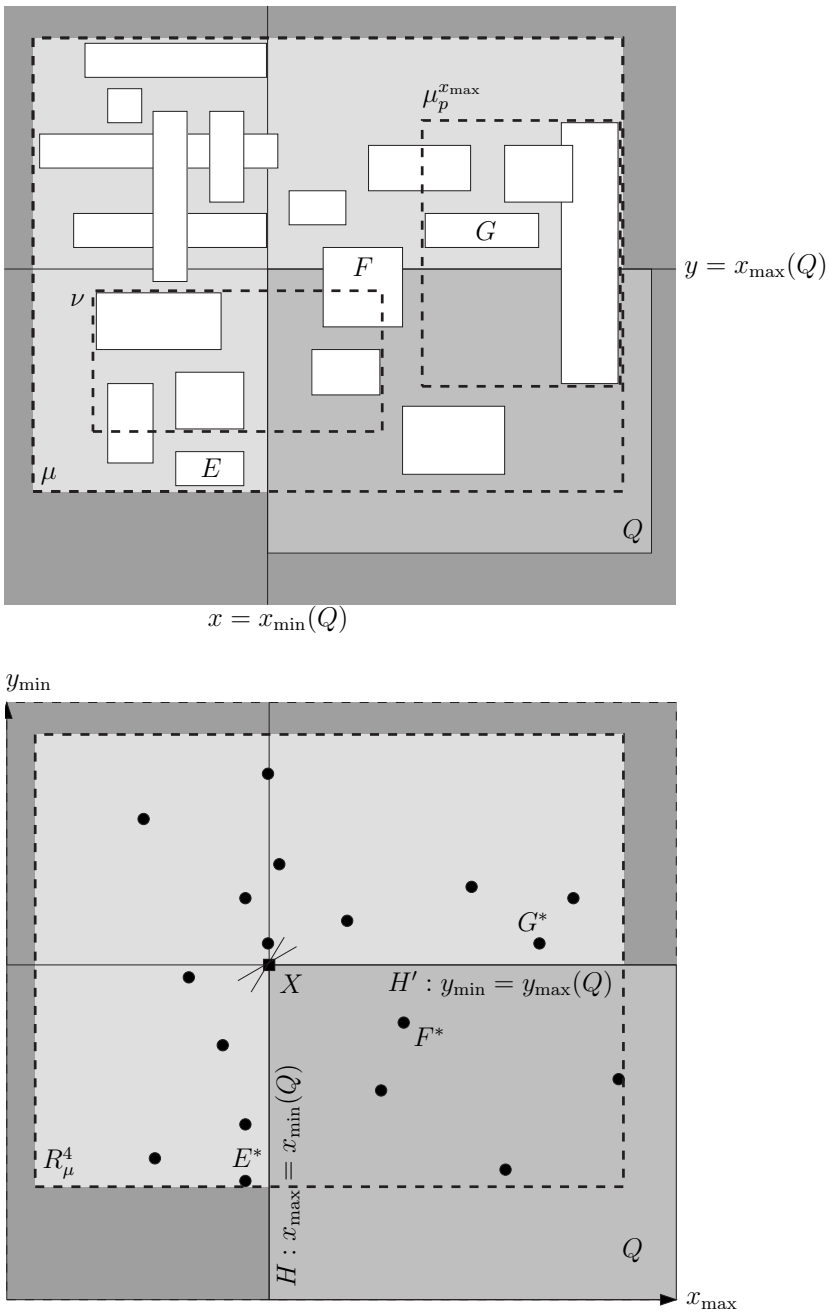

Figure 3: The proof of Lemma 2.2, with μ in the plane (upper figure), and μ in four-dimensional space (lower figure—the $x_{\min}$ and $y_{\max}$ dimensions are not shown). Note that $X = H \cap H'$ is a two-dimensional hyperplane in four-dimensional space. It contains a two-dimensional facet of the transformation of the query range into four dimensions.

By the above arguments, at least two of the three-dimensional hyperplanes defined by $x_{\min}(Q)$, $x_{\max}(Q)$, $y_{\min}(Q)$ and $y_{\max}(Q)$ intersect the region R^4_μ associated with μ when viewing $\mathcal{T}_S$ as a four-dimensional kd-tree. Hence, the intersection X of these two hyperplanes, which is a two-dimensional plane in four-dimensional space, also intersects R^4_μ. With the

priority leaves removed, $\mathcal{T}_S$ becomes a four-dimensional kd-tree with $O(N/B)$ leaves; from a straightforward generalization of the standard analysis of kd-trees we know that any axis-parallel two-dimensional plane intersects at most $O(\sqrt{N/B})$ of the regions associated with the nodes in such a tree [2]. All that remains is to observe that Q defines $O(1)$ such planes, namely one for each pair of sides. Thus $O(\sqrt{N/B})$ is a bound on the number of nodes ν that are not priority leaves and are visited by the query procedure, where not all rectangles in any of the priority leaves below ν's parent are reported. □

2.1.3 Efficient construction algorithm

Note that it is easy to bulk-load a pseudo-PR-tree $\mathcal{T}_S$ on a set S of N rectangles in $O(\frac{N}{B}\log N)$ I/Os by simply constructing one node at a time following the definition in Section 2.1.1. We will now describe how, under the reasonable assumption that the amount M of available main memory is $\Omega(B^{4/3})$, we can bulk-load $\mathcal{T}_S$ using $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os.

Our algorithm is a modified version of the kd-tree construction algorithm described in [1, 20]; it is easiest described as constructing a four-dimensional kd-tree $\mathcal{T}_S$ on the points S^*. In the construction algorithm we first construct, in a preprocessing step, four sorted lists $L_{x_{\min}}$, $L_{y_{\min}}$, $L_{x_{\max}}$, $L_{y_{\max}}$ containing the points in S^* sorted by their $x_{\min}$-, $y_{\min}$-, $x_{\max}$-, and $y_{\max}$-coordinate, respectively. Then we construct $\Theta(\log M)$ levels of the tree, and recursively construct the rest of the tree.

To construct $\Theta(\log M)$ levels of $\mathcal{T}_S$ efficiently we proceed as follows. We first choose a parameter z (which will be explained below) and use the four sorted lists to find the (kN/z)-th coordinate of the points S^* in each dimension, for all $k \in \{1, 2, ..., z-1\}$. These coordinates define a four-dimensional grid of size z^4; we then scan S^* and count the number of points in each grid cell. We choose z to be $\Theta(M^{1/4})$, so that we can keep these counts in main memory.

Next we build the $\Theta(\log M)$ levels of $\mathcal{T}_S$ without worrying about the priority leaves: To construct the root ν of $\mathcal{T}_S$, we first find the slice of z^3 grid cells with common $x_{\min}$-coordinate such that there is a hyperplane orthogonal to the $x_{\min}$-axis that passes through these cells and has at most half of the points in S^* on one side and at most half of the points on the other side. By scanning the $O(N/(Bz))$ blocks from $L_{x_{\min}}$ that contain the $O(N/z)$ points in these grid cells, we can determine the exact $x_{\min}$-value x to use in ν such that the hyperplane H, defined by $x_{\min} = x$, divides the points in S^* into two subsets with at most half of the points each. After constructing ν, we subdivide the z^3 grid cells intersected by H, that is, we divide each of the z^3 cells in two at x and compute their counts by rescanning the $O(N/(Bz))$ blocks from $L_{x_{\min}}$ that contain the $O(N/z)$ points in these grid cells. Then we construct a kd-tree on each side of the hyperplane defined by x recursively (cycling through all four possible cutting directions). Since we create $O(z^3)$ new cells every time we create a node, we can ensure that the grid still fits in main memory after constructing z nodes, that is, $\log z = \Theta(\log M)$ levels of $\mathcal{T}_S$.

After constructing the $\Theta(\log M)$ kd-tree levels, we construct the four priority leaves for each of the z nodes. To do so we reserve main memory space for the B points in each of the priority leaves; we have enough main memory to hold all priority leaves, since by the assumption that M is $\Omega(B^{4/3})$ we have $4 \cdot \Theta(B) \cdot \Theta(z) = O(M)$. Then we fill the priority leaves by scanning S^* and "filtering" each point R_i^* through the kd-tree, one by one, as follows: We start at the root of ν of $\mathcal{T}_S$, and check its priority leaves $\nu_p^{x_{\min}}$, $\nu_p^{y_{\min}}$, $\nu_p^{x_{\max}}$, and $\nu_p^{y_{\max}}$ one by one in that order. If we encounter a non-full leaf we simply place R_i^* there; if we encounter a full leaf ν_p and R_i^* is more extreme in the relevant direction than the least extreme point

R_j^* in ν_p, we replace R_j^* with R_i^* and continue the filtering process with R_j^*. After checking $\nu_p^{y_{\max}}$ we continue to check the priority leaves of the child of ν in $\mathcal{T}_S$ whose region contains the point we are processing; if ν does not have such a child (because we arrived at leaf level in the kd-tree) we simply continue with the next point in S^*.

It is easy to see that the above process correctly constructs the top $\Theta(\log M)$ levels of the pseudo-PR-tree $\mathcal{T}_S$ on S, except that the kd-tree divisions are slightly different than the ones defined in Section 2.1.1, since the points in the priority leaves are not removed before the divisions are computed. However, the bound of Lemma 2.2 still holds: The $O(T/B)$ term does not depend on the choice of the divisions, and the kd-tree analysis that brought the $O(\sqrt{N/B})$ term only depends on the fact that each child gets at most half of the points of its parent.

After constructing the $\Theta(\log M)$ levels and their priority leaves, we scan through the four sorted lists $L_{x_{\min}}$, $L_{y_{\min}}$, $L_{x_{\max}}$, $L_{y_{\max}}$ and divide them into four sorted lists for each of the $\Theta(z)$ leaves of the constructed kd-tree, while omitting the points already stored in priority leaves. These lists contain $O(N/z)$ points each; after writing the constructed kd-tree and priority leaves to disk we use them to construct the rest of $\mathcal{T}_S$ recursively.

Note that once the number of points in a recursive call gets smaller than M, we can simply construct the rest of the tree in internal memory one node at a time. This way we can make slightly unbalanced divisions, so that we have a multiple of B points on one side of each dividing hyperplane. Thus we can guarantee that we get at most one non-full leaf per subtree of size $\Theta(M)$, and obtain almost 100% space utilization. To avoid having an underfull leaf that may violate assumptions made by update algorithms, we may make the priority leaves under its parent slightly smaller so that all leaves contain $\Theta(B)$ rectangles. This also implies that the bound of Lemma 2.1 still holds.

Lemma 2.3 *A pseudo-PR-tree can be bulk-loaded with N rectangles in the plane in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os.*

Proof: The initial construction of the sorted lists takes $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os. To construct $\Theta(\log M)$ levels of $\mathcal{T}_S$ we use $O(N/B)$ I/Os to construct the initial grid, as well as $O(N/(Bz))$ to construct each of the z nodes for a total of $O(N/B)$ I/Os. Constructing the priority leaves by filtering also takes $O(N/B)$ I/Os, and so does the distribution of the remaining points in S^* to the recursive calls. Thus each recursive step takes $O(N/B)$ I/Os in total. The lemma follows since there are $O(\log\frac{N}{B}/\log M) = O(\log_M\frac{N}{B})$ levels of recursion. □

2.2 Two-dimensional PR-tree

In this section we describe how to obtain a PR-tree (with degree $\Theta(B)$ and all leaves on the same level) from a pseudo-PR-tree (with degree six and leaves on all levels), while maintaining the $O(\sqrt{N/B}+T/B)$ I/O window query bound.

The PR-tree is built in stages bottom-up: In stage 0 we construct the leaves V_0 of the tree from the set $S_0 = S$ of N input rectangles; in stage $i \geqslant 1$ we construct the nodes V_i on level i of the tree from a set S_i of $O(N/B^i)$ rectangles, consisting of the minimal bounding boxes of all nodes in V_{i-1} (on level $i-1$). Stage i consists of constructing a pseudo-PR-tree $\mathcal{T}_{S_i}$ on S_i; V_i then simply consists of the (priority as well as normal) leaves of $\mathcal{T}_{S_i}$; the internal nodes

are discarded.[3] The bottom-up construction ends when the set S_i is small enough so that the rectangles in S_i and the pointers to the corresponding subtrees fit into one block, which is then the root of the PR-tree.

Theorem 2.4 *A PR-tree on a set S of N rectangles in the plane can be bulk-loaded in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os, such that a window query can be answered in $O(\sqrt{N/B}+T/B)$ I/Os.*

Proof: By Lemma 2.3, stage i of the PR-tree bulk-loading algorithm uses $O((|S_i|/B)\log_{M/B}(|S_i|/B)) = O((N/B^{i+1})\log_{M/B}\frac{N}{B})$ I/Os. Thus the complete PR-tree is constructed in

$$\sum_{i=0}^{O(\log_B N)} O\left(\frac{N}{B^{i+1}}\log_{M/B}\frac{N}{B}\right) = O\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right) \text{I/Os.}$$

To analyze the number of I/Os used to answer a window query Q, we will analyze the number of nodes visited on each level of the tree. Let T_i ($i \leqslant 0$) be the number of nodes visited on level i. Since the nodes on level 0 (the leaves) correspond to the leaves of a pseudo-PR-tree on the N input rectangles S, it follows from Lemma 2.2 that $T_0 = O(\sqrt{N/B}+T/B)$; in particular, there are constants N' and c such that for $N/B \geqslant N'$, $B \geqslant 4c^2$, and $T \geqslant 0$, we have $T_0 \leqslant c\sqrt{N/B} + c(T/B)$. There must be T_{i-1} rectangles in nodes of level $i \geqslant 1$ of the PR-tree that intersect Q, since these nodes contain the bounding boxes of nodes on level $i-1$. Since nodes on level i correspond to the leaves of a pseudo-PR-tree on the N/B^i rectangles in S_i, it follows from Lemma 2.2 that for $N/B^{i+1} \geqslant N'$ and $B \geqslant 4c^2$ we have $T_i \leqslant (c/\sqrt{B^i})\sqrt{N/B} + c(T_{i-1}/B)$. We can now write out the recurrence for $N/B^{i+1} \geqslant N'$, that is, for $i \leqslant (\log_B \frac{N}{N'}) - 1$:

$$\begin{aligned}
T_i &\leqslant \frac{c}{\sqrt{B^i}}\sqrt{\frac{N}{B}} + c\frac{T_{i-1}}{B} \\
&\leqslant \frac{c}{\sqrt{B^i}}\left(1+\frac{c}{\sqrt{B}}\right)\sqrt{\frac{N}{B}} + c\frac{c}{B}\frac{T_{i-2}}{B} \\
&\leqslant \ldots \\
&\leqslant \frac{c}{\sqrt{B^i}}\left(\sum_{j=0}^{i}\left(\frac{c}{\sqrt{B}}\right)^j\right)\sqrt{\frac{N}{B}} + c\left(\frac{c}{B}\right)^i\frac{T}{B}
\end{aligned}$$

With $B \geqslant 4c^2$, it follows that:

$$T_i \leqslant \frac{2c}{(2c)^i}\sqrt{\frac{N}{B}} + \frac{c}{(4c)^i}\frac{T}{B}$$

Summing over all levels $i \leqslant (\log_B \frac{N}{N'}) - 1$, we find that the total number of nodes visited on those levels is at most:

$$\sum_{i=0}^{\lfloor \log_B \frac{N}{N'} \rfloor - 1} \frac{2c}{(2c)^i}\sqrt{\frac{N}{B}} + \frac{c}{(4c)^i}\frac{T}{B} = O\left(\sqrt{\frac{N}{B}} + \frac{T}{B}\right)$$

[3]There is a subtle difference between the pseudo-PR-tree algorithm used in stage 0 and the algorithm used in stages $i > 0$. In stage 0, we construct leaves with input rectangles. In stages $i > 0$, we construct nodes with pointers to children and bounding boxes of their subtrees. The number of children that fits in a node might differ by a constant factor from the number B of rectangles that fits in a leaf, so the number of children might be $\Theta(B)$ rather than B. For our analysis the difference does not matter and is therefore ignored for simplicity.

The higher levels, with less than N' nodes, just add an additive constant, so we conclude that $O(\sqrt{N/B} + T/B)$ nodes are visited in total. □

2.3 Multi-dimensional PR-tree

In this section we briefly sketch how our PR-tree generalizes to dimensions greater than two. We focus on how to generalize pseudo-PR-trees, since a d-dimensional PR-tree can be obtained using d-dimensional pseudo-PR-trees in exactly the same way as in the two-dimensional case; that the d-dimensional PR-tree has the same asymptotic performance as the d-dimensional pseudo-PR-tree is also proved exactly as in the two-dimensional case.

Recall that a two-dimensional pseudo-PR-tree is basically a four-dimensional kd-tree, where four priority leaves containing extreme rectangles in each of the four directions have been added below each internal node. Similarly, a d-dimensional pseudo-PR-tree is basically a $2d$-dimensional kd-tree, where each node has $2d$ priority leaves with extreme rectangles in each of the $2d$ standard directions. For constant d, the structure can be constructed in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os using the same grid method as in the two-dimensional case (Section 2.1.3); the only difference is that in order to fit the $2d$-dimensional grid in main memory we have to decrease z (the number of nodes produced in one recursive stage) to $\Theta(M^{1/2d})$.

To analyze the number of I/Os used to answer a window query on a d-dimensional pseudo-PR-tree, we analyze the number of visited internal nodes as in the two-dimensional case (Section 2.1.2); the total number of visited nodes is at most a factor $2d$ higher, since at most $2d$ priority leaves can be visited per internal node visited. As in the two-dimensional case, $O(T/B)$ is a bound on the number of nodes ν visited where all rectangles in at least one of the priority leaves below ν's parent are reported. The number of nodes ν visited such that each priority leaf of ν's parent contains at least one rectangle not intersecting the query can then be bounded using an argument similar to the one used in two dimensions; it is equal to the number of regions associated with the nodes in a $2d$-dimensional kd-tree with $O(N/B)$ leaves that intersect the $(2d-2)$-dimensional intersection of two orthogonal hyperplanes. It follows from a straightforward generalization of the standard kd-tree analysis that this is $O((N/B)^{1-1/d})$ [2].

Theorem 2.5 *A PR-tree on a set of N d-dimensional hyper-rectangles can be bulk-loaded in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os, such that a window query can be answered in $O((N/B)^{1-1/d} + T/B)$ I/Os.*

2.4 LPR-tree: doing insertions and deletions

In this section we describe and analyze the logarithmic pseudo-PR-tree, or LPR-tree for short. This tree enables us to maintain an R-tree-like structure efficiently without losing the worst-case optimal query time. The structure of an LPR-tree differs from a normal R-tree in two ways. First, the leaves are on different levels. Second, the internal nodes store some additional information, which is explained below. Still, exactly the same query algorithms as for a real R-tree can be used on an LPR-tree. We describe the LPR-tree in two dimensions—generalization to higher dimensions can be done in the same way as with PR-trees.

2.4.1 Structure

An LPR-tree consists of a root with a number of subtrees. Each subtree is a normal pseudo-PR-tree, except that the internal nodes (kd-nodes) store some additional information, and the kd-nodes are grouped to share blocks on disk. Both adaptations serve to make efficient deletions possible. We will refer to these subtrees as APR-trees (annotated pseudo-PR-trees).

In each internal node ν of an APR-tree, the following information is stored:

- pointers to all of ν's children, and a bounding box for each child;
- the split value which was used to cut ν in the four-dimensional kd-tree;
- for each priority leaf of ν, the least extreme value of the relevant coordinate of any rectangle stored in that leaf, that is:
  - the largest $x_{\min}$-coordinate in $\nu_p^{x_{\min}}$;
  - the largest $y_{\min}$-coordinate in $\nu_p^{y_{\min}}$;
  - the smallest $x_{\max}$-coordinate in $\nu_p^{x_{\max}}$;
  - the smallest $y_{\max}$-coordinate in $\nu_p^{y_{\max}}$.

Recall that the internal nodes of pseudo-PR-trees have degree only six. In an APR-tree, we group these nodes into blocks as follows. With each internal node at depth i in a tree such that $i = 0 \pmod{\lfloor \log B \rfloor}$, we store, in the same block, all its descendant internal nodes down to level $i + \lfloor \log B \rfloor - 1$. In the following sections, we will keep writing about nodes that share a block as separate nodes, but in the analysis, we will use the fact that we can follow any path of length l down into the tree with only $O(l/\log B)$ I/Os.

An LPR-tree is the structure that results from applying the logarithmic method [4, 20] to APR-trees. An LPR-tree has up to $\lceil \log(N/B) \rceil + 3$ subtrees $\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_{\lceil \log(N/B) \rceil + 2}$. Subtree $\mathcal{T}_0$ stores at most B rectangles, and $\mathcal{T}_i$ $(i > 0)$ is an APR-tree that stores at most $2^{i-1}B$ rectangles. So $\mathcal{T}_0$ and $\mathcal{T}_1$ will never contain more than one leaf; the other subtrees may have more nodes. The smaller subtrees, that is $\mathcal{T}_0$ up to tree $\mathcal{T}_m$ for some $m = \log \frac{M}{B} - O(1)$, have a total size of at most $B + \sum_{i=0}^{\log(M/B)-O(1)} 2^{i-1}B = O(M)$; we keep these subtrees completely in main memory. From the larger subtrees, that is tree $\mathcal{T}_{\lceil \log(N/B) \rceil + 2}$ down to tree $\mathcal{T}_l$, for some $l = \log \frac{N}{M} + O(1)$, we keep the top $i - l$ levels in main memory; these have a total size of $\sum_{i=l}^{\lceil \log(N/B) \rceil + 2} O(2^{i-l}B) = O(\sum_{i=0}^{\log(N/B) - \log(N/M)} 2^i B) = O(M)$. The lower levels of the larger subtrees are stored on disk. If $l > m + 1$, subtrees $\mathcal{T}_{m+1}, ..., \mathcal{T}_{l-1}$ are stored on disk completely.

2.4.2 Algorithms

To *bulk-load* an LPR-tree with a set of N rectangles, we build an APR-tree on the rectangles and store it as $\mathcal{T}_{\lceil \log N/B \rceil + 1}$. All other subtrees are left empty.

To *insert* a rectangle R in an LPR-tree, we proceed as follows. Check $\mathcal{T}_0$. If $\mathcal{T}_0$ is full, we find the subtree $\mathcal{T}_j$ with smallest j such that $\mathcal{T}_j$ is empty. We take all rectangles from $\mathcal{T}_0$ to $\mathcal{T}_j$ together and build a new APR-tree $\mathcal{T}_j$ on them. The old APR-trees $\mathcal{T}_i$, for $0 \leqslant i < j$, are discarded. Having made sure that there is space in $\mathcal{T}_0$, we add R to $\mathcal{T}_0$.

To *delete* a rectangle R from an LPR-tree, we proceed as follows. We search for R in each subtree $\mathcal{T}_i$. We start at the root of each subtree; in each internal node ν, we compare R to

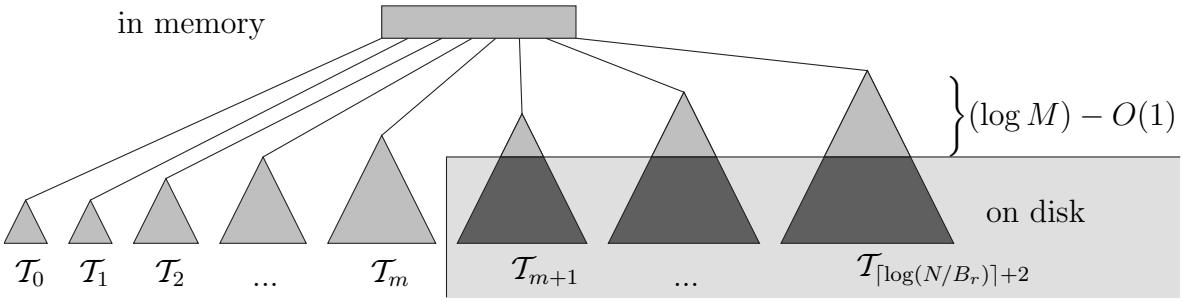

Figure 4: An LPR-tree. The bright part is kept in main memory; the dark part is stored on disk.

the information stored about the priority leaves and the split value of ν to decide in which child to continue the search. When we find the leaf that contains R, we delete R from it. If this leaf is a priority leaf ν_p and its parent ν has one or two kd-nodes (λ and possibly μ) as children, we check if ν_p still contains more than $B/2$ rectangles. If this is the case, we are done. Otherwise, we check out the leaves that follow ν_p in the sequence $\nu_p^{x_{\min}}$, $\nu_p^{y_{\min}}$, $\nu_p^{x_{\max}}$, $\nu_p^{y_{\max}}$, the priority leaves of λ (or λ itself, if it is a leaf), and the priority leaves of μ (or μ itself), to find the $B/2$ rectangles in those leaves that are the most extreme in the relevant coordinate. We move those rectangles into ν_p. As a result, one or more of the leaves drawn from may have become underfull, that is: containing $B/2$ rectangles or less; we will replenish them in a similar manner. Leaves that are kd-nodes, and priority leaves of kd-nodes that do not have kd-children, cannot be replenished. We will just leave them underfull, and delete them when they become completely empty.

Every now and then we do a *clean-up*, where we rebuild the entire LPR-tree from scratch (using the bulk-loading algorithm). More precisely, we maintain a counter N_0, which is the number of rectangles present at the last clean-up[4], a counter I, which is the number of insertions since then, and a counter D, which is the number of deletions since then. As soon as $D \geqslant N_0/2$ or $I \geqslant N_0$, we do a clean-up.

2.4.3 Query complexity

Let us first verify that we can query an LPR-tree efficiently.

Lemma 2.6 *A window query in an LPR-tree on N rectangles in the plane needs $O(\sqrt{N/B} + T/B)$ I/Os in the worst-case.*

Proof: An APR-tree has a structure very similar to a pseudo-PR-tree, but as a result of deletions, an APR-tree can be somewhat unbalanced. Nevertheless, even after deletions, the kd-nodes in an APR-tree $\mathcal{T}_i$ ($i > 0$) will always form a subset of the kd-nodes of an APR-tree on at most $2^{i-1}B$ rectangles. Furthermore, the deletion algorithm ensures that the priority leaves in an APR-tree always contain $\Theta(B)$ rectangles, except possibly the priority leaves of kd-nodes that have no kd-children. It is easy to see now that the analysis for pseudo-PR-trees (proof of Lemma 2.2) still goes through, if we just write 2^{i-1} rather than $O(N/B)$ for

[4]When the LPR-tree is initialized by bulk-loading, we set N_0 to the number of rectangles present at bulk-loading, and consider the bulk-loading to be the first clean-up.

the number of leaves. Thus we find that the number of nodes visited in an APR-tree $\mathcal{T}_i$ is $O(\sqrt{2^{i-1}} + T_i/B)$, where T_i is the number of answers found in $\mathcal{T}_i$.

Taking the sum of the number of I/Os needed for all trees $\mathcal{T}_i$, we find:

$$\sum_{i=m+1}^{\lceil \log \frac{N}{B} \rceil + 2} O\left(\sqrt{2^{i-1}} + \frac{T_i}{B}\right) = O\left(\sqrt{\frac{N}{B}} + \frac{T}{B}\right)$$

□

2.4.4 Bulk-loading complexity

Lemma 2.7 *An LPR-tree can be bulk-loaded with $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os.*

Proof: For bulk-loading the APR-tree $\mathcal{T}_{\lceil \log N/B \rceil + 1}$, we use to same algorithm as for pseudo-PR-trees, now storing the additional information and grouping the internal nodes into blocks at no additional I/O-cost. The algorithm uses
$O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os (Lemma 2.3). □

2.4.5 Insertion complexity

Lemma 2.8 *Inserting a rectangle in an LPR-tree takes*
$O(\frac{1}{B}(\log_{M/B} \frac{N}{B})(\log_2 \frac{N}{M}))$ I/Os amortized.

Proof: We will bound the I/Os spent on the insertions done in between any two clean-up operations, including the I/Os needed for the second clean-up if that was caused by an insertion.

Just after clean-up, N_0 rectangles are present, and they are all stored in $\mathcal{T}_{k+1}$, with $k = \lceil \log N_0/B \rceil$. Recall that an insertion, if $\mathcal{T}_0$ is full, finds the first empty tree $\mathcal{T}_j$, and then constructs $\mathcal{T}_j$ from $\mathcal{T}_1, \ldots, \mathcal{T}_{j-1}$. Tree $\mathcal{T}_{k+1}$ would only become involved in this when a new rectangle is to be inserted after $2^k B$ insertions have filled up all trees $\mathcal{T}_i$ with $0 \leqslant i \leqslant k$. However, this cannot happen before the second clean-up, since a clean-up is done as soon as $N_0 < 2^k B$ rectangles have been inserted—or even earlier, if it is triggered by deletions. If the clean-up is caused by an insertion, we charge the cost, $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os (Lemma 2.7), to the $N_0 = \Theta(N)$ insertions that caused it, which is $O(\frac{1}{B} \log_{M/B} \frac{N}{B})$ I/Os amortized per insertion. (Clean-ups caused by deletions will be charged to the deletions—see Section 2.4.6.)

It remains to account for the construction of trees $\mathcal{T}_j$ with $m \leqslant j \leqslant k$ in between clean-ups (trees $\mathcal{T}_j$ with $0 \leqslant m$ are constructed in main memory). Note that only rectangles inserted since the last clean-up are involved in this. By Lemma 2.7, the cost of constructing $\mathcal{T}_j$ is $O(\frac{1}{B} \log_{M/B} \frac{N}{B})$ I/Os amortized per rectangle in the tree. When $\mathcal{T}_j$ is constructed, all rectangles that are put in come from a tree $\mathcal{T}_i$ with $i < j$. It follows that a rectangle can be included at most $k - m + 1 = O(\log_2 \frac{N}{M})$ times in a new tree that is (partly) built in external memory. This leads to an amortized cost of moving rectangles between clean-ups of $O(\frac{1}{B}(\log_{M/B} \frac{N}{B})(\log_2 \frac{N}{M}))$ I/Os.

Adding the bounds, we see that moving rectangles between clean-ups dominates, and gives the bound claimed. □

2.4.6 Deletion complexity

Lemma 2.9 *Deleting a rectangle from an LPR-tree takes* $O((\log_B \frac{N}{M})(\log_2 \frac{N}{M}))$ *I/Os amortized.*

Proof: We first need to find the rectangle. In the worst case, we have to check all APR-trees, $O(\log_2 \frac{N}{M})$ of which are (partially) stored on disk. Since the higher levels of these trees are stored in main memory, and the remaining internal nodes are blocked in groups of height $\Theta(\log_B)$, walking down a path in such an APR-tree takes at most $O(\log_B \frac{N}{M})$ I/Os. In total, $O((\log_B \frac{N}{M})(\log_2 \frac{N}{M}))$ I/Os may be needed to locate the rectangle.

The replenishing of the priority leaves is accounted for as follows. Let the *external height* h of a priority leaf ν_p be the largest number of kd-nodes that are found on any path from ν down into the APR-tree and are stored on disk, or have their priority leaves stored on disk. Since the higher levels of the larger APR-trees are stored in main memory, $h = O(\log_2 \frac{N}{M})$. Let the *rank* of a priority leaf be four times its external height, minus its rank among its siblings, i.e. -1 for $\nu_p^{x_{\min}}$; -2 for $\nu_p^{y_{\min}}$; -3 for $\nu_p^{x_{\max}}$, and -4 for $\nu_p^{y_{\max}}$. When we remove a rectangle from a priority leaf, we put a charge of $2r/B$ in its place, where r is the rank of the priority leaf. We only replenish a priority leaf if it gets half-empty, which implies that it contains a total charge of r. By moving rectangles from lower-ranked priority leaves in, we create gaps in those priority leaves, but since all of these have lower rank, we need to put a total charge of at most $r-1$ in their place. Hence, the replenishing of a priority leaf ν_p frees a charge of at least 1, which pays for the $O(1)$ I/Os that are needed to replenish ν_p. Replenishing priority leaves thus takes $O(\frac{\max(r)}{B}) = O(\frac{1}{B}\log_2 \frac{N}{M})$ I/Os amortized per deletion.

When D becomes $N_0/2$, which is more than $N/4$, the LPR-tree is rebuilt at a cost of $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os (Lemma 2.7). This is $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ per deletion, amortized.

Adding the amortized cost of locating and deleting the rectangle, replenishing the priority leaves and rebuilding the LPR-tree, we find that a deletion takes $O((\log_B \frac{N}{M})(\log_2 \frac{N}{M}))$ I/Os amortized. □

2.4.7 Speeding up deletions

When the insertion of a rectangle leads to building a subtree $\mathcal{T}_i$, we put in all rectangles that were inserted by then and are not stored in a previously built subtree $\mathcal{T}_j$ with $j > i$. This makes it possible to find a rectangle in an LPR-tree without searching all $\log_2 \frac{N}{M}$ subtrees T_i that are stored on disk: we just need to keep track of the time of insertion of each rectangle. When we want to find a particular rectangle in the LPR-tree, we only need to search the subtree T_i that was constructed earliest after the rectangle's insertion in the LPR-tree.

To make this work, we need to keep three additional structures with the LPR-tree:

- a single number *time*, initially set to zero, which we increase with every update so that we can use it to put unique time stamps on update operations;
- a *time index*, implemented as a B-tree of which the top $\Theta(\log_B M)$ levels are kept in main memory; the lower levels are stored on disk. The time index stores for every rectangle in the forest the time stamp of its insertion. It can do so using any type of key that uniquely identifies rectangles.
- in main memory: for each non-empty subtree $\mathcal{T}_i$, the time at which it was built.

The algorithms are modified as follows. When an LPR-tree is cleaned up, we set *time* to zero, and rebuild the time index to store the zero time stamp for all rectangles. When a subtree $\mathcal{T}_i$ is built or modified because of an insertion, we increment *time* and record it as the time of construction of $\mathcal{T}_i$. When a rectangle is inserted, we also increment *time* and insert the rectangle's key with this time stamp in the time index. When a rectangle R is deleted, we query the time index to get the rectangle's time of insertion, find the subtree $\mathcal{T}_i$ that was constructed earliest after the rectangle was inserted, and search only $\mathcal{T}_i$ to find R.

Lemma 2.10 *An LPR-tree with time stamps can be bulk-loaded with* $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ *I/Os.*

Proof: The LPR-tree itself is built with $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os (Lemma 2.7). The time index can be built in the same time bound: sort the keys of the rectangles in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os and build a B-tree on them. □

Lemma 2.11 *Inserting a rectangle in an LPR-tree with time stamps takes* $O((\log_B\frac{N}{M}) + \frac{1}{B}(\log_{M/B}\frac{N}{B})(\log_2\frac{N}{M}))$ *I/Os amortized.*

Proof: Inserting the rectangle in the LPR-tree takes$O(\frac{1}{B}(\log_{M/B}\frac{N}{B})(\log_2\frac{N}{M}))$ I/Os amortized (Lemma 2.8). Inserting the rectangle in the time index takes $O(\log_B\frac{N}{M})$ I/Os. Add these bounds to get the bound claimed. □

Lemma 2.12 *Deleting a rectangle from an LPR-tree with time stamps takes* $O(\log_B\frac{N}{M})$ *I/Os amortized.*

Proof: We first find the rectangle in the time index; this takes $O(\log_B\frac{N}{M})$ I/Os. With the result we determine which subtree $\mathcal{T}_i$ to search; walking down that subtree takes at most $O(\log_B\frac{N}{M})$ I/Os. Replenishing the priority leaves takes $O(\frac{1}{B}\log_2\frac{N}{M})$ I/Os amortized (see the proof of Lemma 2.9), and rebuilding the LPR-tree as soon as $D \geqslant N_0/2$ takes $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os (Lemma 2.9), that is $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ I/Os amortized per deletion. Adding it all up, we find that a deletion takes $O(\log_B\frac{N}{M})$ I/Os amortized. □

2.4.8 LPR-tree: the bounds

The lemmas above lead to the following theorem.

Theorem 2.13 *An LPR-tree with time stamps on a set of N rectangles in the plane can be bulk-loaded in* $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ *I/Os, such that a window query can be answered in* $O(\sqrt{N/B} + T/B)$ *I/Os in the worst case, a rectangle can be inserted in* $O((\log_B\frac{N}{M}) + \frac{1}{B}(\log_{M/B}\frac{N}{B})(\log_2\frac{N}{M}))$ *I/Os amortized, and a rectangle can be deleted in* $O(\log_B\frac{N}{M})$ *I/Os amortized.*

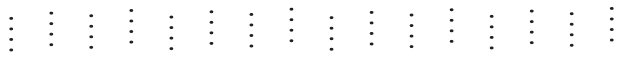

Figure 5: Worst-case example

2.5 Lower bound for heuristic R-trees

The PR-tree is the first R-tree variant that always answers a window query worst-case optimally. In fact, most other R-tree variants can be forced to visit $\Theta(N/B)$ nodes to answer a query even when no rectangles are reported ($T = 0$). In this section we show how this is the case for the packed Hilbert R-tree, the four-dimensional Hilbert R-tree, and the TGS R-tree.

Theorem 2.14 *There exist a set of rectangles S and a window query Q that does not intersect any rectangles in S, such that all $\Theta(N/B)$ nodes are visited when Q is answered using a packed Hilbert R-tree, a four-dimensional Hilbert R-tree, or a TGS R-tree on S.*

Proof: We will construct a set of *points* S such that all leaves in a packed Hilbert R-tree, a four-dimensional Hilbert R-tree, and a TGS R-tree on S are visited when answering a *line* query that does not touch any point. The theorem follows since points and lines are all special rectangles.

For convenience we assume that $B \geq 4$, $N = 2^k B$ and $N/B = B^m$, for some positive integers k and m, so that each leaf of the R-tree contains B rectangles, and each internal node has fanout B. We construct S as a grid of N/B columns and B rows, where each column is shifted up a little, depending on its horizontal position (each row is in fact a Halton-Hammersley point set; see e.g. [8]). More precisely, S has a point $p_{ij} = (x_{ij}, y_{ij})$, for all $i \in \{0, ..., N/B - 1\}$ and $j \in \{0, ..., B - 1\}$, such that $x_{ij} = i + 1/2$, and $y_{ij} = j/B + h(i)/N$. Here $h(i)$ is the number obtained by reversing, i.e. reading backwards, the k-bit binary representation of i. An example with $N = 64, B = 4$ is shown in Figure 5.

Now, let us examine the structure of each of the three R-tree variants on this dataset.

Packed Hilbert R-tree: The packed Hilbert R-tree sorts the points by the Hilbert values. To compare the Hilbert values of two points, we must first check if they lie in the same quadrant of a sufficiently large square whose sides are powers of two: in our case, a square of size 2^k suffices. If they lie in the same quadrant, we zoom in on that quadrant, and see if they lie in the same subquadrant of that quadrant. We keep zooming in until we arrive at the level where the points lie in different quadrants. Then, we decide which quadrant comes first on the Hilbert curve.

Now consider two points p_{ij} and $p_{i'j'}$. Note that both y_{ij} and $y_{i'j'}$ are smaller than 1, so all bits before the "decimal point" of the y-coordinates of these points are the same, namely zero. In addition, if $i = i'$, then $x_{ij} = x_{i'j'}$, and therefore, starting from a square of size 2^k, we have to zoom in more than k levels deep to distinguish between the positions of p_{ij} and $p_{i'j'}$ on the Hilbert curve. On the other hand, if $i \neq i'$, then $|x_{ij} - x_{i'j'}| \geqslant 1$, so the x-coordinates of p_{ij} and $p_{i'j'}$ differ in at least one of the bits before the "decimal" point. Hence, we do not have to zoom in more than k levels and can compare them on the basis of i and i' only, ignoring j and j'. As a result, the Hilbert curve visits the columns in our grid of points one by one, and when it visits a column, it visits all points in that column before proceeding to another column. Therefore, the packed Hilbert R-tree makes a leaf for every column, and a horizontal line can be chosen to intersect all these columns while not touching any point.

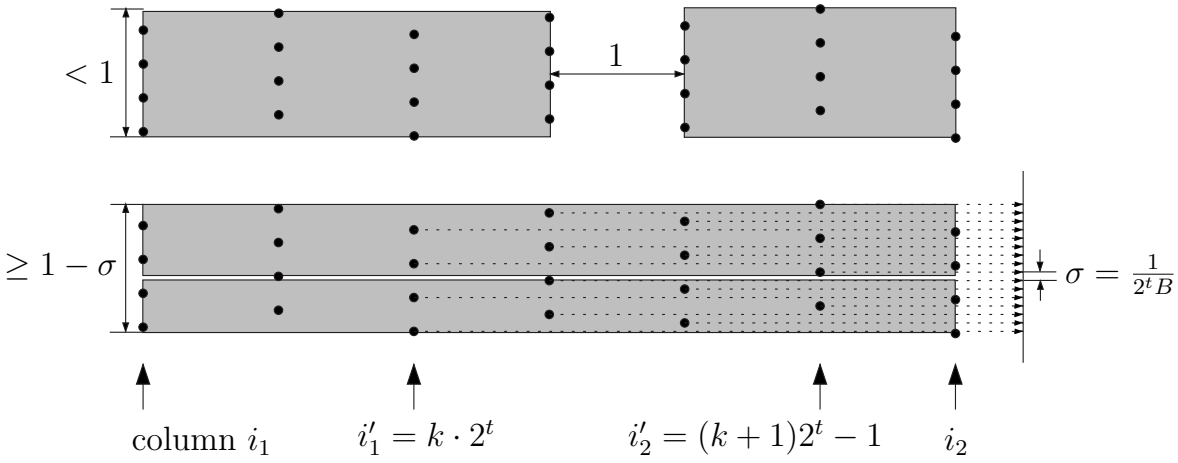

Figure 6: TGS partitioning the worst-case example. A vertical division creates two bounding boxes with a total area of less than $i_2 - i_1 - 1$. A horizontal division creates two bounding boxes with a total area of more than $(i_2 - i_1)(1 - 2\sigma) > i_2 - i_1 - 1$.

Four-dimensional Hilbert R-tree: The analysis is similar to the one for the packed Hilbert R-trees.

TGS R-tree: The TGS algorithm will partition S into B subsets of equal size and partition each subset recursively. The partitioning is implemented by choosing a partitioning line that separates the set into two subsets (whose sizes are multiples of N/B), and then applying binary partitions to the subsets recursively until we have partitioned the set into N subsets of size N/B. Observe that on all levels in this recursion, the partitioning line will leave at least a fraction $1/B$ of the input on each side of the line. Below we prove that TGS will always partition by vertical lines; it follows that TGS will eventually put each column in a leaf. Then a line query can intersect all leaves but report nothing.

Suppose TGS is about to partition the subset $S(i_1, i_2)$ of S that consists of columns i_1 to i_2 inclusive, with $i_2 > i_1$, i.e. $S(i_1, i_2) = \{p_{ij} | i \in \{i_1, ..., i_2\}, j \in \{0, ..., B-1\}\}$. When the greedy split algorithm gets to divide such a set into two, it can look for a vertical partitioning line or for a horizontal partitioning line. Intuitively, TGS favors partitioning lines that create a big gap between the bounding boxes of the points on each side of the line. As we will show below, we have constructed S such that the area of the gap created by a horizontal partitioning line is always roughly the same, as is the area of the gap created by a vertical line, with the latter always being bigger.

Partitioning with a vertical line would always leave a gap of roughly a square that fits between two columns—see Figure 6. More precisely, it would partition the set $S(i_1, i_2)$ into two sets $S(i_1, c-1)$ and $S(c, i_2)$, for some $c \in \{i_1+1, ..., i_2\}$. The bounding boxes of these two sets would each have height less than 1, and their total width would be $(c-1-i_1)+(i_2-c)$, so their total area A_v would be less than $i_2 - i_1 - 1$.

The width of a gap around a horizontal partitioning line depends on the number of columns in $S(i_1, i_2)$. However, the more columns are involved the bigger the density of the points in those columns when projected on the y-axis, and the lower the gap that can be created—see Figure 6 for an illustration. As a result, partitioning with a horizontal line can lead to gaps that are wide and low, or relatively high but not so wide; in any case, the area of the gap will

be roughly the same. More precisely, we can estimate the total area of the bounding boxes resulting from partitioning with a horizontal line as follows. The partitioning line must leave at least a fraction $1/B$ of the points on each side, so there must be at least one full row of $S(i_1, i_2)$ on each side of the line. Hence, the width of both bounding boxes resulting from the partition step must be $i_2 - i_1$. Observe that the set $\{i_1, ..., i_2 + 1\}$ contains at least two different multiples of 2^s if s is such that $i_2 + 2 - i_1 \geqslant 2^{s+1}$; let t be the largest such value of s, i.e. $t = \lfloor \log(i_2 + 2 - i_1) - 1 \rfloor$, let i'_1 be the smallest multiple of 2^t that is at least i_1, and let i'_2 be $i'_1 + 2^t - 1$. Note that if we let i go through all values $\{i'_1, ..., i'_2\}$, then the first $k - t$ bits of the k-bit representation of i remain constant, while the last t bits assume all possible values. Consequently, the last $k - t$ bits of $h(i)$ will remain constant, while the first t bits will assume all possible values. Hence, if we project all points in $S(i_1, i_2)$ on the y-axis, the distance between each pair of consecutive points is at most $\sigma = 2^{k-t}/N = 1/(2^t B)$, and the distance between the topmost and the bottommost point is at least $1 - \sigma$. When we partition this set by a horizontal line, the total height of the resulting bounding boxes must be at least $1 - 2\sigma$, and their total area A_h must be at least $(i_2 - i_1)(1 - 2\sigma) = i_2 - i_1 - 2(i_2 - i_1)/(2^t B)$. With $t \geqslant \log(i_2 + 2 - i_1) - 1$, we find that A_h is more than $i_2 - i_1 - 4/B$.

Recall that A_v is less than $i_2 - i_1 - 1$. Since $B \geqslant 4$, we can conclude that $A_h > A_v$, and that partitioning with a vertical line will always result in a smaller total area of bounding boxes than with a horizontal line. As a result, TGS will always cut vertically between the columns. □

3 Experiments

In this section we describe the results of our experimental study of the performance of the PR-tree. We compared the PR-tree to several other bulk-loading methods known to generate query-efficient R-trees: The packed Hilbert R-tree (denoted H in the rest of this section), the four-dimensional Hilbert R-tree (denoted H4), and the TGS R-tree (denoted TGS). Among these, TGS has been reported to have the best query performance, but it also takes many I/Os to bulk-load. In contrast, H is simple to bulk-load, but it has worse query performance because it does not take the extent of the input rectangles into account. H4 has been reported to be inferior to H [15], but since it takes the extent into account (like TGS) it should intuitively be less vulnerable to extreme datasets.

3.1 Experimental setup

We implemented the four bulk-loading algorithms in C++ using TPIE [3]. TPIE is a library that provides support for implementing I/O-efficient algorithms and data structures. In our implementation we used 36 bytes to represent each input rectangle; 8 bytes for each coordinate and 4 bytes to be able to hold a pointer to the original object. Each bounding box in the internal nodes also used 36 bytes; 8 bytes for each coordinate and 4 bytes for a pointer to the disk block storing the root of the corresponding subtree. The disk block size was chosen to be 4KB, resulting in a maximum fanout of 113. This is similar to earlier experimental studies, which typically use block sizes ranging from 1KB to 4KB or fix the fan-out to a number close to 100.

As experimental platform we used a dedicated Dell PowerEdge 2400 workstation with one Pentium III/500MHz processor running FreeBSD 4.3. A local 36GB SCSI disk (IBM

Ultrastar 36LZX) was used to store all necessary files: the input data, the R-trees, as well as temporary files. We restricted the main memory to 128MB and further restricted the amount of memory available to TPIE to 64MB; the rest was reserved to operating system daemons.

3.2 Datasets

We used both real-life and synthetic data in our experiments.

3.2.1 Real-life data

As the real-life data we used the TIGER/Line data [24] of geographical features in the United States. This data is the standard benchmark data used in spatial databases. It is distributed on six CD-ROMs and we chose to experiment with the road line segments from two of the CD-ROMs: disk one containing data for sixteen eastern US states and disk six containing data from five western US states; we use Eastern and Western to refer to these two datasets, respectively. To obtain datasets of varying sizes we divided the Eastern dataset into five regions of roughly equal size, and then put an increasing number of regions together to obtain datasets of increasing sizes. The largest set is just the whole Eastern dataset. For each dataset we used the bounding boxes of the line segments as our input rectangles. As a result, the Eastern dataset had 16.7 million rectangles, for a total size of 574MB, and the Western data set had 12 million rectangles, for a total size of 411MB. Refer to Table 1 for the sizes of each of the smaller Eastern datasets. Note that the biggest dataset is much larger than those used in previous works (which only used up to 100,000 rectangles) [15, 12]. Note also that our TIGER data is relatively nicely distributed; it consist of relatively small rectangles (long roads are divided into short segments) that are somewhat (but not too badly) clustered around urban areas.

dataset:	1	2	3	4	5
million rectangles:	2.08	5.67	9.16	12.66	16.72
size (MB):	72	194	315	435	574

Table 1: The sizes of the Eastern datasets

3.2.2 Synthetic data

To investigate how the different R-trees perform on more extreme datasets than the TIGER data, we generated a number of synthetic datasets. Each of these synthetic datasets consisted of 10 million rectangles (or 360MB) in the unit square.

- SIZE(max_side): We designed the first class of synthetic datasets to investigate how well the R-trees handle rectangles of different sizes. In the SIZE(max_side) dataset the rectangle centers were uniformly distributed and the lengths of their sides uniformly and independently distributed between 0 and max_side. When generating the datasets, we discarded rectangles that were not completely inside the unit square (but made sure each dataset had 10 million rectangles). A portion of the dataset SIZE(0.001) is shown in Figure 7.

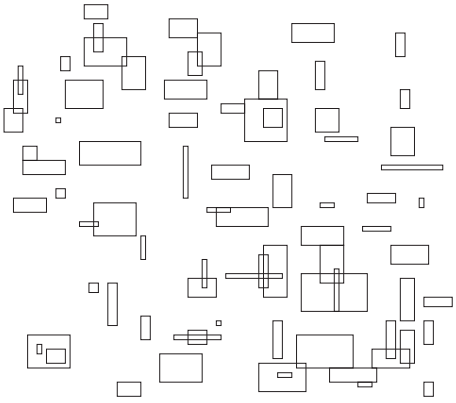

Figure 7: Synthetic dataset SIZE(0.001)

- ASPECT(a): The second class of synthetic datasets was designed to investigate how the R-trees handle rectangles with different aspect ratios. The areas of the rectangles in all the datasets were fixed to 10^{-6}, a reasonably small size. In the ASPECT(a) dataset the rectangle centers were uniformly distributed but their aspect ratios were fixed to a and the longest sides chosen to be vertical or horizontal with equal probability. We also made sure that all rectangles fell completely inside the unit square. A portion of the dataset ASPECT(10) is shown in Figure 8. Note that if the input rectangles are bounding boxes of line segments that are almost horizontal or vertical, one will indeed get rectangles with very high aspect ratio—even infinite in the case of horizontal or vertical segments.

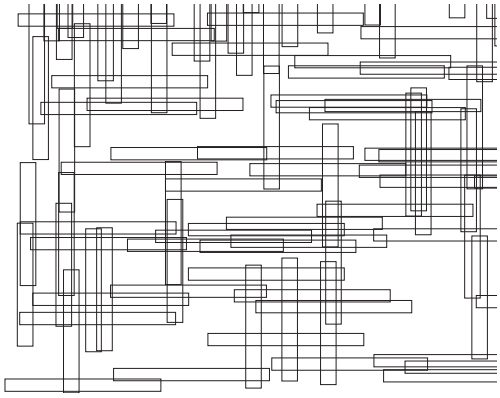

Figure 8: Synthetic dataset ASPECT(10)

- SKEWED(c): In many real-life multidimensional datasets different dimensions often have different distributions. Some of these distributions may be highly skewed compared to the others. We designed the third class of datasets to investigate how this affects R-tree performance. SKEWED(c) consists of uniformly distributed points that have been "squeezed" in the y-dimension, that is, each point (x, y) is replaced with (x, y^c). An example of SKEWED(5) is shown in Figure 9.

- CLUSTER: Our final dataset was designed to illustrate the worst-case behavior of the H,

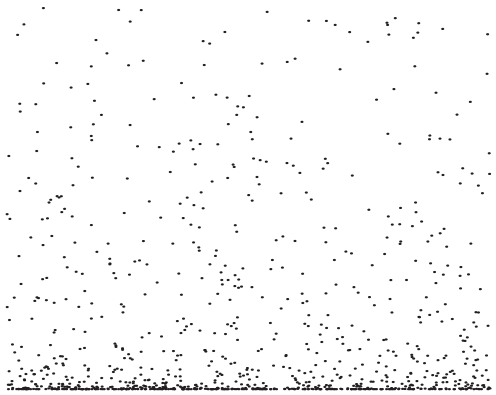

Figure 9: Synthetic dataset SKEWED(5)

H4 and TGS R-trees. It is similar to the worst-case example discussed in Section 2. It consists of 10 000 clusters with centers equally spaced on a horizontal line. Each cluster consists of 1000 points uniformly distributed in a $0.000\,01 \times 0.000\,01$ square surrounding its center. Figure 10 shows a part of the CLUSTER dataset.

Figure 10: Synthetic dataset CLUSTER

3.3 Experimental results

Below we discuss the results of our bulk-loading and query experiments with the four R-tree variants.

3.3.1 Bulk-loading performance

We bulk-loaded each of the R-trees with each of the real-life TIGER datasets, as well as with the synthetic datasets for various parameter values. In all experiments and for all R-trees we achieved a space utilization above 99%.[5] We measured the time spent and counted the number of 4KB blocks read or written when bulk-loading the trees. Note that all algorithms we tested read and write blocks almost exclusively by sequential I/O of large parts of the data; as a result, I/O is much faster than if blocks were read and written in random order.

Figure 11 shows the results of our experiments using the Eastern and Western datasets. Both experiments yield the same result: The H and H4 algorithms use the same number of I/Os, and roughly 2.5 times fewer I/Os than PR. This is not surprising since even though the three algorithms have the same $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/O bounds, the PR algorithm is much more complicated than the H and H4 algorithms. The TGS algorithm uses roughly 4.5 times more I/Os than PR, which is also not surprising given that the algorithm makes binary partitions

[5] When R-trees are bulk-loaded to subsequently be updated dynamically, near 100% space utilization is often not desirable [10]. However, since we are mainly interested in the query performance of the R-tree constructed with the different bulk-loading methods, and since the methods could be modified in the same way to produce non-full leaves, we only considered the near 100% utilization case.

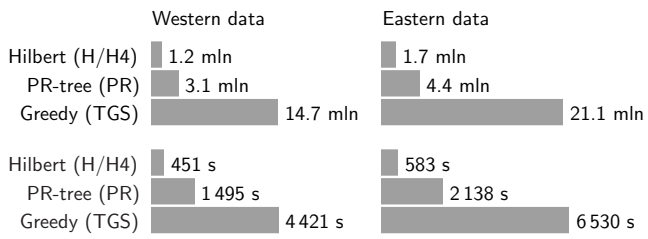

Figure 11: Bulk-loading performance on TIGER data: I/O (upper figure) and time (lower figure).

so that the number of levels of recursion is effectively $O(\log_2 N)$. In terms of time, the H and H4 algorithms are still more than 3 times faster than the PR algorithm, but the TGS algorithm is only roughly 3 times slower than PR. This shows that H, H4 and PR are all more CPU-intensive than TGS.

Figure 12 shows the results of our experiments with the five Eastern datasets. These experiments show that the H, H4 and PR algorithms scale relatively linearly with dataset size; this is a result of the $\lceil \log_{M/B} \frac{N}{B} \rceil$ factor in the bulk-loading bound being the same for all datasets. For H and H4 this means that the core step of the algorithm, which is sorting the rectangles by their position on the Hilbert curve, runs in the same number of passes—namely two—in all experiments. For the PR algorithm it means that only once, we have to build a grid and divide among its cells a set of rectangles that is too big too fit in memory. All recursive steps can be done in main memory. The cost of the TGS algorithm seems to grow in an only slightly superlinear way with the size of the data set. This is a result of the $\lceil \log_B N \rceil$ factor in the bulk-loading bound being almost the same for all data sets. It means that in all data sets, the subtrees just below root level have roughly the same size ($B^{\lceil \log_B N \rceil - 1}$ rectangles), and since each of them is too big to fit in main memory, we need a significant number of I/Os to build them. This, together with preprocessing, accounts for a big portion of the number of I/Os that scales linearly with the size of the data set. The slightly superlinear trend comes from the cost of building the root, which varies from roughly 20% of the total number of I/Os on set 1, to roughly 45% on set 5 (shaded in Figure 12).

In our experiments with the synthetic data we found that the performance of the H, H4 and PR bulk-loading algorithms was practically the same for all the datasets, that is, unaffected by the data distribution. This is not surprising, since the performance should only depend on the dataset size (and all the synthetic datasets have the same size). The PR algorithm performance varied slightly, which can be explained by the small effect the data distribution can have on the grid method used in the bulk-loading algorithm (subtrees may have slightly different sizes due to the removal of priority boxes). On average, the H and H4 algorithms spent 381 seconds and 1.0 million I/Os on each of the synthetic datasets, while the PR algorithm spent 1289 seconds and 2.6 million I/Os. On the other hand, as expected, the performance of the TGS algorithm varied significantly over the synthetic datasets we tried; the binary partitions made by the algorithm depend heavily on the input data distribution. The TGS algorithm was between 4.6 and 16.4 times slower than the PR algorithm in terms of I/O, and between 2.8 and 10.9 times slower in terms of time. The performance of TGS on the SIZE(max_side), ASPECT(a), SKEWED(c) and CLUSTER datasets is shown in Figure 13.

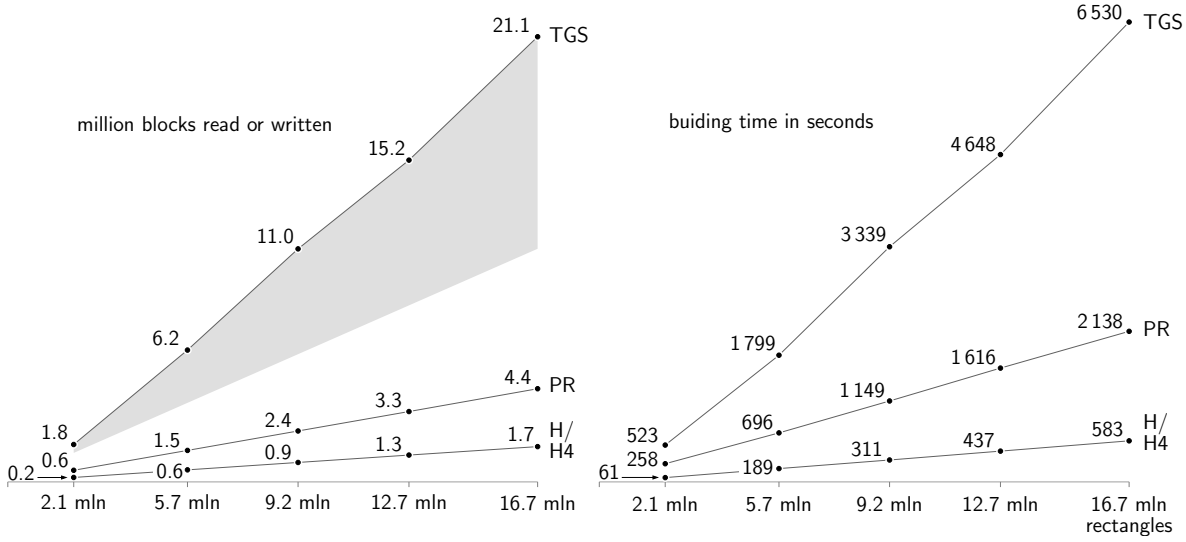

Figure 12: Bulk-loading performances on Eastern datasets: I/O (left) and time (right)

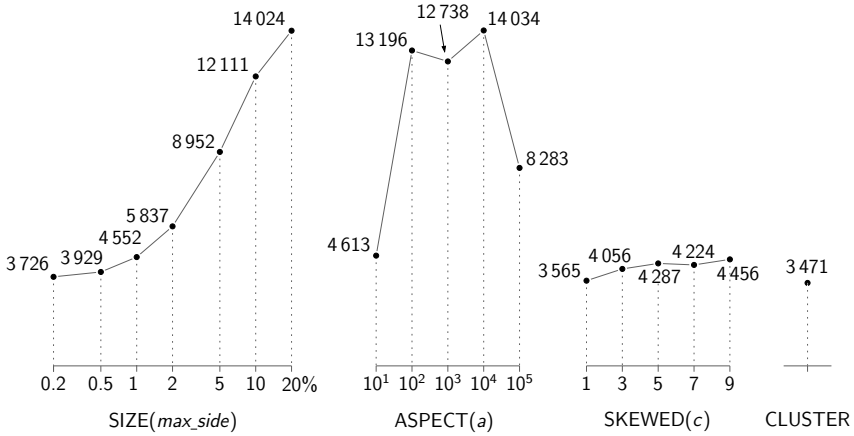

Figure 13: Bulk-loading time in seconds of Top-down Greedy Split on synthetic data sets of 10 million rectangles (SIZE and ASPECT) or points (SKEWED and CLUSTER) each.

3.3.2 Query performance

After bulk-loading the four R-tree variants we experimented with their query performance; in each of our experiments we performed 100 randomly generated queries and computed their average performance (a more exact description of the queries is given below). Following previous experimental studies, we utilized a cache (or "buffer") to store internal R-tree nodes during queries. In fact, in all our experiments we cached all internal nodes since they never occupied more than 6MB. This means that when reporting the number of I/Os needed to answer a query, we are in effect reporting the number of leaves visited in order to answer the

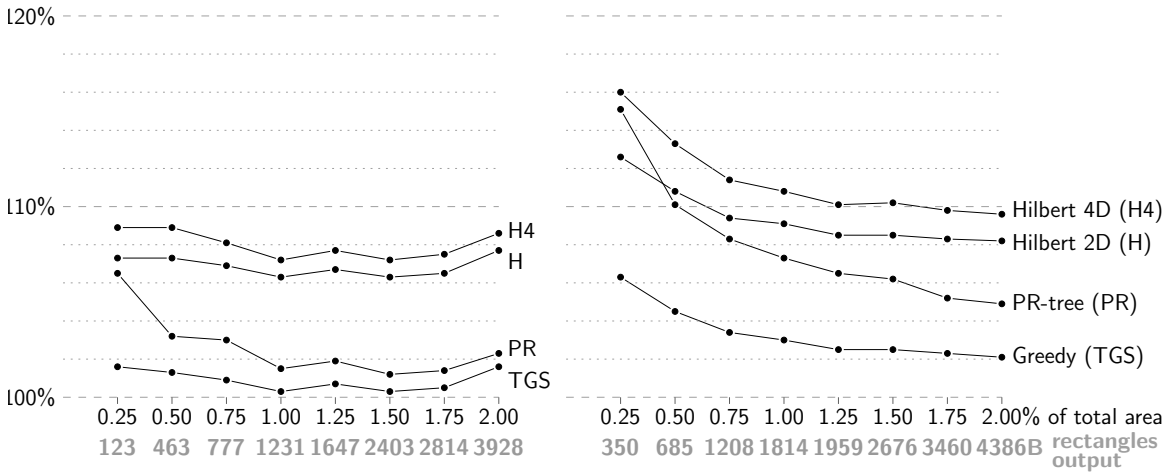

Figure 14: Query performance for queries with squares of varying size on the Western TIGER data (left) and the Eastern TIGER data (right). The performance is given as the number of blocks read divided by the output size T/B.

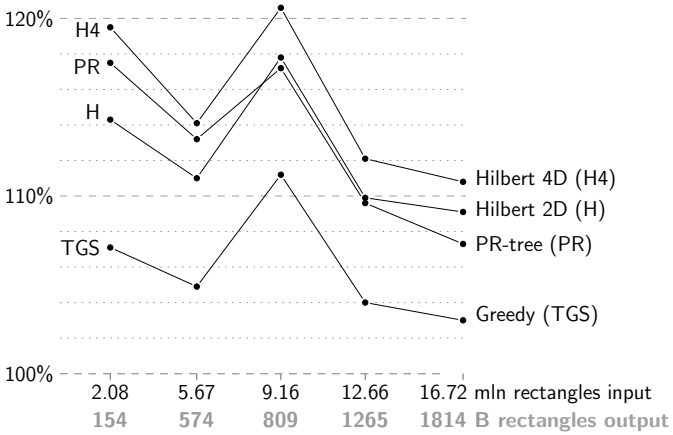

Figure 15: Query performance for queries with squares of area 0.01 on Eastern TIGER data sets of varying size. The performance is given as the number of blocks read divided by the output size T/B.

query.[6] For several reasons, and following previous experimental studies [6, 12, 15, 16], we did not collect timing data. Two main reasons for this are (1) that I/O is a much more robust measure of performance, since the query time is easily affected by operating system caching and by disk block layout; and (2) that we are interested in heavy load scenarios where not much cache memory is available or where caches are ineffective, that is, where I/O dominates the query time.

[6]Experiments with the cache disabled showed that in our experiments the cache actually had relatively little effect on the window query performance.

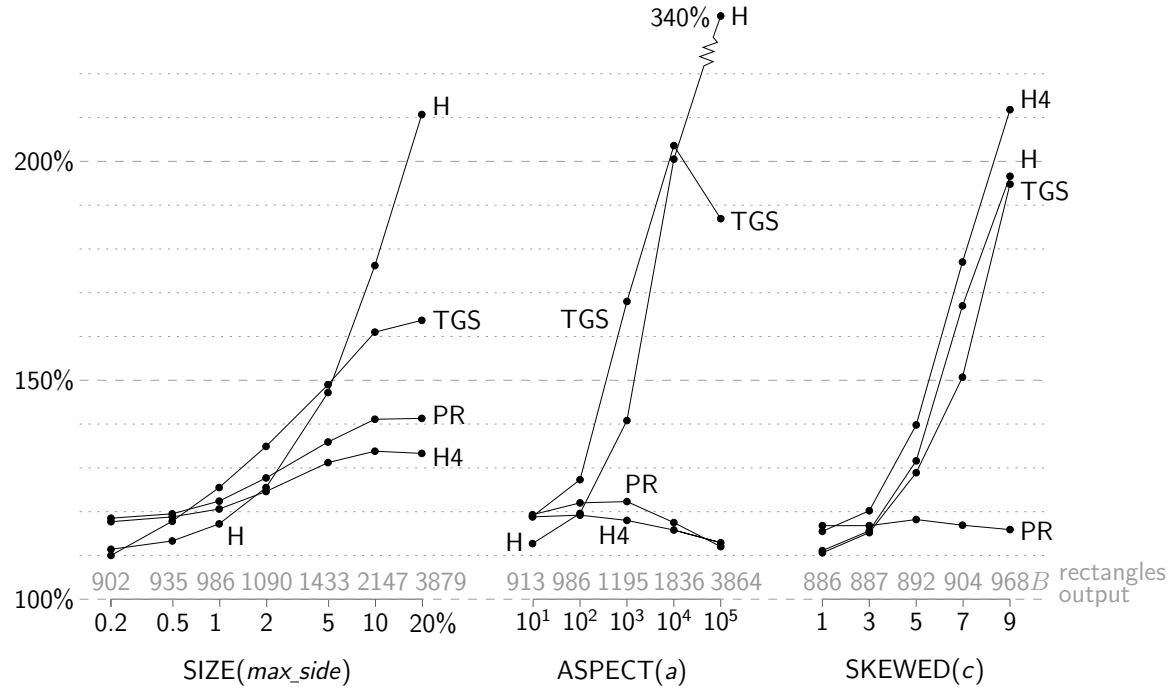

Figure 16: Query performance for queries with squares of area 0.01 on synthetic data sets. The performance is given as the number of blocks read divided by the output size T/B.

TIGER data: We first performed query experiments using the Eastern and Western datasets. The results are summarized in Figure 14 and 15. In Figure 14 we show the results of experiments with square window queries with areas that range from 0.25% to 2% of the area of the bounding box of all input rectangles. We used smaller queries than previous experimental studies (for example, the maximum query in [15] occupies 25% of the area) because our datasets are much larger than the datasets used in previous experiments—without reducing the query size the output would be unrealistically large and the reporting cost would thus dominate the overall query performance. In Figure 15 we show the results of experiments on the five Eastern datasets of various sizes with a fixed query size of 1%. The results show that all four R-tree variants perform remarkably well on the TIGER data; their performance is within 10% of each other and they all answer queries in close to T/B, the minimum number of necessary I/Os. Their relative performance generally agrees with earlier results [15, 12], that is, TGS performs better than H, which in turn is better than H4. PR consistently performs slightly better than both H and H4 but slightly worse than TGS.

Synthetic data. Next we performed experiments with our synthetic datasets, designed to investigate how the different R-trees perform on more extreme data than the TIGER data. For each of the datasets SIZE, ASPECT and SKEWED we performed experiments where we varied the parameter to obtain data ranging from fairly normal to rather extreme. Below we summarize our results.

The left side of Figure 16 shows the results of our experiments with the dataset SIZE(max_side) when varying max_side from 0.002 to 0.2, that is, from relatively small to relatively large rectangles. As queries we used squares with area 0.01. Our results show that for relatively small input rectangles, like the TIGER data, all the R-tree variants perform very close to the

minimum number of necessary I/Os. However, as the input rectangles get larger, PR and H4 clearly outperform H and TGS. H performs the worst, which is not surprising since it does not take the extent of the input rectangles into account. TGS performs significantly better than H but still worse than PR and H4. Intuitively, PR and H4 can handle large rectangles better, because they rigorously divide rectangles into groups of rectangles that are similar in all four coordinates. This may enable these algorithms to group likely answers, namely large rectangles, together so that they can be retrieved with few I/Os. It also enables these algorithms to group small rectangles nicely, while TGS, which strives to minimize the total area of bounding boxes, may be indifferent to the distribution of the small rectangles in the presence of large rectangles.

The middle of Figure 16 shows the results of our experiments with the dataset ASPECT(a), when we vary a from 10 to 10^5, that is, when we go from rectangles (of constant area) with small to large aspect ratio. As query we again used squares with area 0.01. The results are very similar to the results of the SIZE dataset experiments, except that as the aspect ratio increases, PR and H4 become significantly better than TGS and especially H. Unlike with the SIZE dataset, PR performs as well as H4 and they both perform close to the minimum number of necessary I/Os to answer a query. Thus this set of experiments re-emphasizes that both the PR-tree and H4-tree are able to adopt to varying extent very well.

The right side of Figure 16 shows the result of our experiments with the dataset SKEWED(c), when we vary c from 1 to 9, that is, when we go from a uniformly distributed point set to a very skewed point set. As query we used squares with area 0.01 that are skewed in the same way as the dataset (that is, where the corner (x, y) is transformed to (x, y^c)) so that the output size remains roughly the same. As expected, the PR performance is unaffected by the transformations, since our bulk-loading algorithm is based only on the relative order of coordinates: x-coordinates are only compared to x-coordinates, and y-coordinates are only compared to y-coordinates; there is no interaction between them. On the other hand, the query performance of the three other R-trees degenerates quickly as the point set gets more skewed.

As a final experiment, we queried the CLUSTER dataset with long skinny horizontal queries (of area 1×10^{-7}) through the 10 000 clusters; the y-coordinate of the leftmost bottom corner was chosen randomly such that the query passed through all clusters. The results are shown in Table 2. As anticipated, the query performance of H, H4 and TGS is very bad; the CLUSTER dataset was constructed to illustrate the worst-case behavior of the structures. Even though a query only returns around 0.3% of the input points on average, the query algorithm visits 37%, 94% and 25% of the leaves in H, H4 and TGS, respectively. In comparison, only 1.2% of the leaves are visited in PR. Thus the PR-tree outperforms the other indexes by well over an order of magnitude.

tree:	H	H4	PR	TGS
# I/Os:	32 920	83 389	1 060	22 158
% of the R-tree visited:	37%	94%	1.2%	25%

Table 2: Query performances on synthetic dataset CLUSTER.

3.4 Conclusions of the experiments

The main conclusion of our experimental study is that the PR-tree is not only theoretically efficient but also practically efficient. Our bulk-loading algorithm is slower than the packed Hilbert and four-dimensional Hilbert bulk-loading algorithms but much faster than the TGS R-tree bulk-loading algorithm. Furthermore, unlike for the TGS R-tree, the performance of our bulk-loading algorithm does not depend on the data distribution. The query performance of all four R-trees is excellent on nicely distributed data, including the real-life TIGER data. On extreme data however, the PR-tree is much more robust than the other R-trees (even though the four-dimensional Hilbert R-tree is also relatively robust).

4 Concluding remarks

In this paper we presented the PR-tree, which is the first R-tree variant that can answer any window query in the optimal $O(\sqrt{N/B} + T/B)$ I/Os. We also performed an extensive experimental study, which showed that the PR-tree is not only optimal in theory, but that it also performs excellent in practice: for normal data, it is quite competitive to the best known heuristics for bulk-loading R-trees, namely the packed Hilbert-R-tree [15] and the TGS R-tree [12], while for data with extreme shapes or distributions, it outperforms them significantly.

The PR-tree can be updated using any known update heuristic for R-trees, but then its performance cannot be guaranteed theoretically anymore and its practical performance might suffer as well. Alternatively, we can use the dynamic version of the PR-tree using the logarithmic method, which has the same theoretical worst-case query performance and can be updated efficiently. In the future we wish to experiment to see what happens to the performance when we apply heuristic update algorithms and when we use the theoretically superior logarithmic method.

References

[1] P. K. Agarwal, L. Arge, O. Procopiuc, and J. S. Vitter. A framework for index bulk loading and dynamization. In *Proc. International Colloquium on Automata, Languages, and Programming*, pages 115–127, 2001.

[2] P. K. Agarwal, M. de Berg, J. Gudmundsson, M. Hammar, and H. J. Haverkort. Box-trees and R-trees with near-optimal query time. *Discrete and Computational Geometry*, 28(3):291–312, 2002.

[3] L. Arge, O. Procopiuc, and J. S. Vitter. Implementing I/O-efficient data structures using TPIE. In *Proc. European Symposium on Algorithms*, pages 88–100, 2002.

[4] L. Arge and J. Vahrenhold. I/O-efficient dynamic planar point location. *International Journal of Computational Geometry & Applications*, 2003. To appear.

[5] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[6] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. SIGMOD International Conference on Management of Data*, pages 322–331, 1990.

[7] S. Berchtold, C. Böhm, and H.-P. Kriegel. Improving the query performance of high-dimensional index structures by bulk load operations. In *Proc. Conference on Extending Database Technology, LNCS 1377*, pages 216–230, 1998.

[8] B. Chazelle. *The Discrepancy Method: Randomness and Complexity.* Cambridge University Press, New York, 2001.

[9] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[10] D. J. DeWitt, N. Kabra, J. Luo, J. M. Patel, and J.-B. Yu. Client-server paradise. In *Proc. International Conference on Very Large Databases*, pages 558–569, 1994.

[11] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[12] Y. J. García, M. A. López, and S. T. Leutenegger. A greedy algorithm for bulk loading R-trees. In *Proc. 6th ACM Symposium on Advances in GIS*, pages 163–164, 1998.

[13] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[14] H. J. Haverkort, M. de Berg, and J. Gudmundsson. Box-trees for collision checking in industrial installations. In *Proc. ACM Symposium on Computational Geometry*, pages 53–62, 2002.

[15] I. Kamel and C. Faloutsos. On packing R-trees. In *Proc. International Conference on Information and Knowledge Management*, pages 490–499, 1993.

[16] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proc. International Conference on Very Large Databases*, pages 500–509, 1994.

[17] K. V. R. Kanth and A. K. Singh. Optimal dynamic range searching in non-replicating index structures. In *Proc. International Conference on Database Theory, LNCS 1540*, pages 257–276, 1999.

[18] S. T. Leutenegger, M. A. López, and J. Edgington. STR: A simple and efficient algorithm for R-tree packing. In *Proc. IEEE International Conference on Data Engineering*, pages 497–506, 1996.

[19] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis. R-trees have grown everywhere. *Technical Report available at http://www.rtreeportal.org/*, 2003

[20] O. Procopiuc, P. K. Agarwal, L. Arge, and J. S. Vitter. Bkd-tree: A dynamic scalable kd-tree. In *Proc. International Symposium on Spatial and Temporal Databases*, 2003.

[21] J. Robinson. The K-D-B tree: A search structure for large multidimensional dynamic indexes. In *Proc. SIGMOD International Conference on Management of Data*, pages 10–18, 1981.

[22] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. In *Proc. SIGMOD International Conference on Management of Data*, pages 17–31, 1985.

[23] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: A dynamic index for multi-dimensional objects. In *Proc. International Conference on Very Large Databases*, pages 507–518, 1987.

[24] *TIGER/Line™ Files, 1997 Technical Documentation.* Washington, DC, September 1998. http://www.census.gov/geo/tiger/TIGER97D.pdf.