University of Denver Computer Science Technical Report # 97-02

A Greedy Algorithm for Bulk Loading R-trees *

Yván J. García Mario A. López Scott T. Leutenegger
Mathematics and Computer Science Department
University of Denver Denver, CO 80208-0189
{*ygarcia,mlopez,leut*}*@cs.du.edu*

Abstract

Offline loading of R-trees is useful to increase node utilization and improve query performance. We present a new algorithm for bulk loading R-trees which differs from previous proposals in at least two important aspects: First, our algorithm partitions input data into subtrees in a top-down fashion as partitions close to the root are likely to have the biggest impact on performance. Second, at each tree level, our algorithm considers all cuts orthogonal to the coordinate axes that result in a balanced tree and greedily selects the ones that optimize a user supplied cost function. Extensive experimental studies with both real and synthetic data indicate that our algorithm is especially competitive for region data, where it results in R-trees that require up to three times fewer disk accesses than previous proposals. More specifically, it is the method of choice for data sets with skew in center point locations, areas, or aspect ratios. Such data sets are common in various application areas.

1 Introduction

A search for spatial objects that satisfy a given multidimensional query is one of the most frequent operations in areas related to spatial databases. Geographic information systems, computer-aided design, computer vision and robotics, temporal and scientific databases, and multi-keyed indexing for traditional databases are examples of such areas. R-trees [3] were developed as an index structure that allows the efficient execution of multidimensional queries. Although this structure is based on (hyper-) rectangles with sides parallel to the axes, arbitrarily shaped objects can be handled by operating with their minimum bounding boxes.

R-trees are dynamic data structures that can be updated incrementally. Efficient insertion and deletion algorithms are provided by Guttman [3]. However, incremental construction using Guttman's algorithms often produces trees with poor space utilization and poor structure (as measured by the number of disk accesses required to satisfy a query). Several efforts to improve the

*This work has been partially supported by the Colorado Advanced Software Institute under grants TT-97-05 and TT-97-06.

quality of incrementally built R-trees have been made[1, 5, 11], but the resulting trees are still inefficient in comparison to R-trees built based on some preprocessing. For data that is known before structuring or not frequently modified, a preprocessing step has been shown to be an effective way of producing R-trees with almost total space utilization and much better query performance. Several "packing algorithms" that perform this type of preprocessing have been proposed in the literature [10, 4, 7].

Roussopoulos' [10] Nearest-X packing algorithm is easy to implement and works fairly well for point queries. Kamel and Faloutsos [4] presented an algorithm that uses a packing order determined by the Hilbert curve and show that query performance (as measured by number of nodes accessed) is proportional to a weighted sum of the areas and perimeters of the minimal rectangles (bounding boxes) that enclose the data of the various subtrees. Their Hilbert algorithm produces a significant improvement over Nearest-X as a result of a reduction in total perimeter while keeping the total area low. Leutenegger, Lopez and Edgington [7] propose Sort-Tile-Recursive, a competitive algorithm that is simple to implement and that often results in significant performance improvements over the previous two algorithms. All of these methods are based on a fairly rigid preprocessing scheme used for a bottom-up construction of the R-tree.

We propose *Top-down Greedy Split* (TGS), a new packing algorithm which builds the tree top-down using an adaptive preprocessing step that aggressively rearranges the database objects under a node to be built. This approach takes advantage of more freedom in the choices (i.e., cuts) to be made during preprocessing, and attacks the performance issue beginning with the nodes where more "dead space" is likely to be removed, i.e., the ones close to the root (due to the fact that these nodes are responsible for larger areas).

Actual data sets from GIS and VLSI design as well as synthetic data sets were used for the performance comparison of the R-trees built by Sort-Tile-Recursive, Hilbert and Top-down-Greedy-Split. Our algorithm produces R-trees that require up to three times fewer disk accesses than trees built by other methods. However, due to the multiple arrangements considered, it also requires higher loading times than those of previous proposals. This is not a problem if the data does not need to be preprocessed frequently (such as maps or static data stored on CD). TGS is particularly effective for data sets with skewed distributions of area, aspect ratio, or center point locations. Our analysis of realistic data from various application areas shows that this type of profile is common in practice.

In [6] Leutenegger and Lopez study how buffering affects performance of the R-tree and show that ignoring the role of the buffer may lead to wrong conclusions when comparing various R-tree construction algorithms. Accordingly, experimental results were gathered through actual implementations with an LRU buffer as explained in Section 3.

The remainder of the paper is organized as follows. Background information on R-trees and descriptions of the packing algorithms used as reference for performance comparison are presented in Section 2. We introduce our proposed algorithm with its analysis in Section 3. Section 4 details the experimental methodology we use for gathering the results discussed in Section 5. We conclude with Section 6.

2 The R-tree and Packing Algorithms

What follows is a basic overview of the R-tree and the two packing algorithms Hilbert [4] and Sort-Tile-Recursive (STR) [7] used as reference for our studies. The understanding of the rest of this paper does not require full knowledge of these algorithms.

2.1 R-trees

Developed for efficient processing of intersection queries on spatial databases, an R-tree is a tree-based index similar to the B^+-tree. R-trees keep a set of (hyper-) rectangles and allows the handling of arbitrarily shaped objects by representing each one with its *minimum bounding rectangle* (MBR). Since the R-tree extension to more than two dimensions is straight forward, we explicitly work only with the 2-dimensional case.

An R-tree node is allowed to hold at most m (R,P) pairs. If the node is a leaf, R is the minimum bounding rectangle (MBR) of the actual database object pointed to by P. Otherwise, R is the MBR of all rectangles stored in the subtree pointed to by P. Note that: (a) rectangles at any level may overlap, (b) an R-tree created from a particular set of objects is not guaranteed to be unique and (c) every descending path in the tree is a sequence of nested rectangles with the last one containing an actual database object.

A query region Q is executed by retrieving and examining each rectangle that intersects Q at every level. A recursive procedure descending the tree from the root through possibly several paths suffices for this retrieval. To process a node, out of the rectangles it stores, those intersecting Q

are retrieved. For non-leaf nodes, a recursive search is requested on child nodes of the retrieved rectangles. The retrieved rectangles of the leaf nodes are simply reported. For the rest of the paper we assume that exactly one node fits per disk page and the terms node and page will be used interchangeably.

2.2 Packing Algorithms

We denote by m the maximum number of rectangles per node, and by r the number of rectangles in the data file. Hilbert and STR packing follow a general process similar to building a B-tree from a set of keys by creating the leaf level first and then creating each next higher level up to the root [9].

Usual Packing Algorithm Framework:

1. Preprocess the data file so that the r rectangles are ordered in $\lceil r/m \rceil$ consecutive groups of m rectangles, where each group of m is to be placed in the same node. The last group may contain fewer than m rectangles.
2. Load the $\lceil r/m \rceil$ groups of rectangles into pages and output the (MBR, page-number) for each page into a temporary file. The page-numbers are used as the child pointers in the nodes of the next higher level.
3. Recursively pack these MBRs into nodes at the next level, proceeding upwards, until the root node is created.

The Hilbert and STR algorithms differ only in how the rectangles are ordered at each level. It should be noted that, in contrast to Hilbert and STR, our algorithm (TGS) builds the R-tree in a top-down fashion.

Hilbert Sort (HS): For the algorithm of [4] the ordering is imposed by the distance along the Hilbert curve from the origin of the curve to the center of each rectangle. The algorithm takes advantage of the fact that points that are close along the Hilbert curve are also close in the original multidimensional space (although the converse is not true). Kamel and Faloutsos only discuss how to handle integer coordinates. A simple way to extend their method to arbitrary floating point values is described in [7].

Sort-Tile-Recursive (STR): For a k-dimensional set of r (hyper-) rectangles, the ordering imposed by the STR algorithm [7] is based on the "tiling" of the plane for the 2D case (and generalized to higher dimension) using $\sqrt{r/m}$ vertical slices so that each slice contains enough rectangles to pack roughly $\sqrt{r/m}$ nodes. Determine the number of leaf level pages $P = \lceil r/m \rceil$ and let $S = \lceil \sqrt{P} \rceil$. Sort the rectangles by its center's x-coordinate and partition them into S vertical slices. A slice consists of a run of $S \cdot m$ consecutive rectangles from the sorted list (the last may have fewer rectangles). Now sort the rectangles of each slice by its center's y-coordinate and pack them into nodes by grouping them into runs of length m

For the case $k > 2$, first, sort the (hyper-) rectangles according to the first coordinate of their center. Then divide the input set into $S = \lceil P^{\frac{1}{k}} \rceil$ slabs, where a slab consists of a run of $m \cdot S^{k-1}$ consecutive (hyper-) rectangles from the sorted list[1]. Each slab is now processed recursively using the remaining $k-1$ coordinates (i.e., treated as a $k-1$-dimensional data set).

3 The Top-down Greedy Splitting Algorithm (TGS)

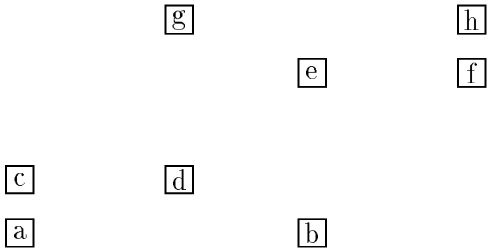
We propose TGS as an alternative that builds the R-tree top-down while using an aggressive approach that greedily constructs the various subtrees of the R-tree. The description of TGS below is followed by a careful analysis of its loading time based on the average number of disk accesses required to build the R-tree.

3.1 Description of TGS

Two key ideas were used for the development of this algorithm:

1. An optimal R-tree cannot, in general, be obtained by minimizing the tree cost, one level at at time. In practice, packing by successively processing each level independently makes the problem tractable and often results in reasonable trees. In contrast to traditional approaches that apply this process in a bottom-up fashion, we first attempt to minimize the cost of the levels that allow a potentially bigger reduction in the overall cost, i.e., the top levels of the R-tree. Note that for the example provided in Figure 1 over all the possible R-trees with $m = 2$, the one with minimum cost (as measured by sum of areas of all MBRs) does not correspond to the one with minimum bottom level area. In fact, the latter has a cost which is 1.6 times bigger than the cost of the best top-down constructed tree.

[1] The expression $m \cdot S^{k-1}$ was misprinted in [7].



	lower left	upper right
a	0 , 0	1 , 1
b	52 , 0	53 , 1
c	0 , 2	1 , 3
d	47 , 2	48 , 3
e	52 , 15	53 , 15
f	99 , 15	100 , 15
g	47 , 17	48 , 18
h	99 , 17	100 , 18

Figure 1: The total min. area 2168 corresponds to the packing (((ab)(cd))((ef)(gh))), but minimizing area level in a bottom up fashion gives a total area of 3542, for the packing (((ac)(be))((dg)(fh))).

2. Previous algorithms use a fairly rigid approach for deciding how to partition a given set of rectangles. As a result they often miss characteristics of the input that may significantly affect performance. Instead, we consider a much wider set of splitting possibilities which results in an algorithm that is "more adaptive" to the geometry of the input rectangles. More specifically, we evaluate all partitions induced by hyper-planes orthogonal to a coordinate axis that result in a balanced and packed tree[2].

TGS is based on the repeated application of a *basic split step*, which takes a set of n rectangles and partitions it into two sets by using a cut orthogonal to an axis such that the sum of costs of the two ensuing parts is minimum. Candidate cuts are constrained so that one side of the cut has a number of rectangles which is a multiple of a parameter M chosen so as to guarantee that the resulting subtrees are packed (thus assuring same node utilization as previous algorithms). The process is then applied to both parts, independently, until the target number of partitions is obtained. (This number can be computed from n and m.)

Consider a set of n rectangles that need to be split. Let $M = m^{(\lceil \log_m n \rceil - 1)}$ be the number of rectangles in each child subtree and $f(r_1, r_2)$ a user-supplied function that measures the cost of a split by adding the costs of the MBRs of rectangles at either side of the split. For each dimension, at least one ordering of the input rectangles is derived. Initial orderings are produced by sorting the input data before the very first invocation of basic split. (The orderings we considered are based on

[2] A packed tree is one where at most one node per level has fewer than m entries.

min. coord., max. coord., both, and center of the input rectangles.)

A description of the basic split process follows:

1. if $n \leq M$ stop
2. for each dimension d

   for each ordering considered in this dimension

   for i from 1 to $\lceil n/M \rceil - 1$

   remember i if a cut after position $M \cdot i$ (in the current order) results in a better value of $f(B_0, B_1)$, where B_0 (resp. B_1) is the MBR of the set of rectangles before (resp. after and including) the current position.
3. Split the input set (and the lists reflecting the orderings in consideration) at the best found position. (At this point TGS applies the basic split to each resulting part.)

Several variants of the basic approach above are possible:

(a) Use only one of { min. coord., max. coord.}, instead of both, for the possible orderings of each dimension.

(b) Consider the center's point coordinate as the only possible ordering.

(c) Use both ascending and descending ordering when n is not a multiple of M.

(d) Construct the tree level by level (i.e., breadth-first) by keeping M as a constant until the current level is done, or depth first by replacing Step 1 with: if $n <= M$ then $M = M/m$ and stop only if M results in 1.

The tree resulting from (a) offers a query performance which is worse by a factor of 2, whereas trees built by (c) show a negligible difference (less than 1%). The results for variant (b) will be discussed later. Variants derived from (d) build the same tree using different approaches (so, there is no performance difference). The MBRs of the leaf level nodes from the R-trees created by each of three bulk loading algorithms for GIS data for Long Beach County are shown in Figures 2, 3 and 4.

3.2 Loading Time Analysis

We now analyze the disk access requirements of TGS which we express in terms of average number of accesses per page. Accesses to disk in TGS originate from two sources:

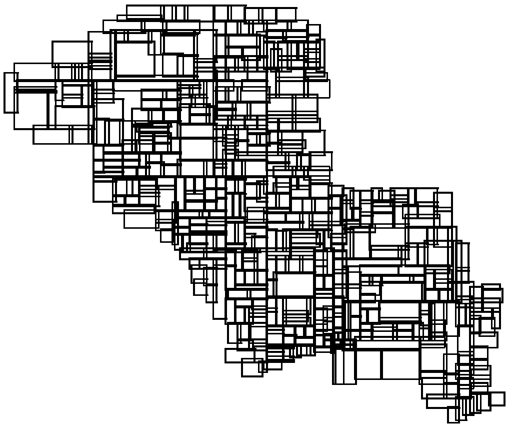


Figure 2: leaf level MBRs for TGS's R-tree

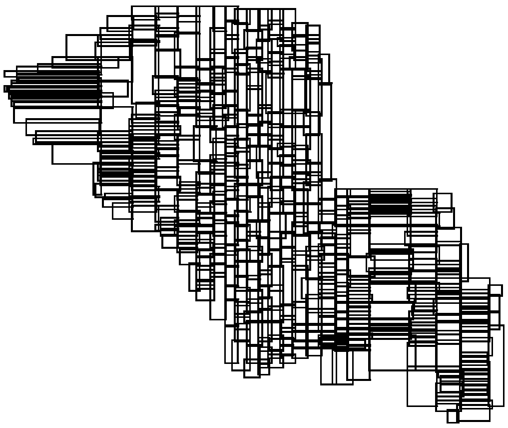


Figure 3: leaf level MBRs for STR's R-tree

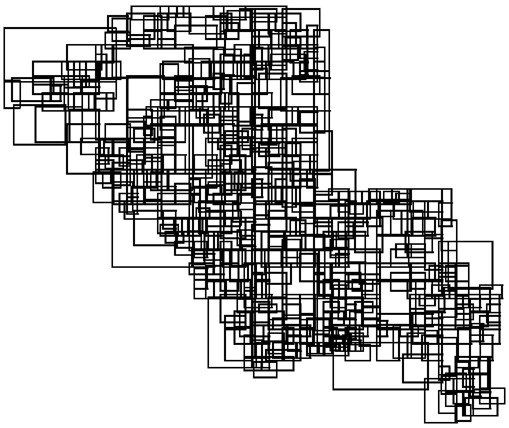


Figure 4: leaf level MBRs for Hilbert's R-tree

1. initial sorting required to prepare the c orderings used, and

2. additional accesses due to repeated application of the basic split process.

Consider first the disk access requirements due to sorting. We assume that one instance of external sorting can be done in four accesses per page[3]. Since c orderings are required it would appear that $4 \cdot c$ accesses per page are needed. However, every time a page is initially read it can be sorted in memory and output c times in the correct orders. Accordingly, initial sorting requires a total of $3 \cdot c + 1$ accesses per page. Since we evaluate one or two orderings per dimension, $c \leq 2 \cdot d$.

We now analyze the requirements of step (2) above. Assuming that $O(m \cdot \log_m r)$ main memory is available, we first show that two accesses per page suffice to split an ordering into two parts (see Step 3 of the basic split process). Then, we use this fact to compute the total number of accesses for the rectangle set involved in an R-tree node, and finally, for the whole tree. Note that for realistic data the height of the R-tree, $\log_m r$, is a very small integer.

As the list of rectangles associated with a node is being output, construct in memory an array B of size at most m (only $\lceil n/M \rceil \leq m$ entries are actually needed), where $B[i]$ contains the MBR of the i-th group of M rectangles written to disk. Using B we can compute the best cut efficiently in main memory as it allows us to operate on a small number of clusters of rectangles with the property that every cut necessarily occurs between clusters.

Once B is available in memory, there are several ways to compute B_0, B_1 for each candidate position. The following three methods offer a trade-off between memory and running time:

1. With no additional memory, for a total $O(m^2)$ time. For all the m positions to consider, at each i, go through the array with incremental building of B_0 from entry 1 to entry i, and then incrementally build B_1 from $i+1$ up to m.

2. In time $O(m \log m)$ with an additional array of $\log m$ entries. Partition B from left to right into groups whose size is a power of 2 (in ascending order of group size) and compute the MBR for each group. A group of size 2^j is created provided that the jth bit of the binary representation of m is 1. At each iteration the smallest group of size 2^k is broken into groups of size $2^0, 2^0, 2^1, \ldots, 2^{k-1}$ (unless k=0) and their MBRs are recomputed. B_0 is updated in $O(1)$ time by merging it with the first singleton group. B_1 is computed in $O(\log m)$ time from the

[3] Each page is read twice and also written twice. In the sequel we do not distinguish between read and write access.

MBRs of the $O(\log m)$ groups. Since a group of size 2^k is broken $n/2^{k+1}$ times the total bound of $O(m \log m)$ time follows.

3. In time $O(m)$, with an additional array D of m entries. incrementally build $D[j]$ as the MBR of B entries from m down to j by traversing B in descending order. Once D is full, while traversing B in an ascending order, B_0 is computed as the incremental MBR of entries 1 up to i. B_1 is simply $D[i+1]$.

Each of the orderings (except for the one that gave the best choice) is split by reading its records in order and writing them into one of the two parts imposed by the partition. During the split, two B arrays are created (one per part). In a depth-first construction, space for at most one additional B per R-tree level is needed (to take care of the second basic split instance). Thus, $O(m \cdot \log_m r)$ memory is enough to hold all pending work.

Consider now the set of n input rectangles associated with a particular R-tree node. To simplify the analysis assume initially that $d = 2$ and $c = 2$. A cluster (i.e., the rectangles that produced a B entry) is accessed whenever a list (associated with an ordering) is split at the best cut. As discussed above this requires 2 accesses per page.

Consider the binary tree whose internal nodes represent a recursive call of the basic split step and whose leaves represent the different clusters (at most m) involved in the construction of an R-tree node. Clearly, this is a full tree, i.e., a tree where each internal node has exactly two children. Note that the number of basic split steps in which a cluster participates is equal to its depth. Thus, ignoring for the moment the accesses due to sorting at the beginning of TGS, the number of accesses per page for a single cluster is equal to twice its depth. Accordingly, the average number of accesses per page over the n rectangles is just twice the sum of all leaf depths divided by the number of leaves (at most m).

Since the full binary tree which maximizes the sum of leaf depths is one with at most 2 nodes per level, the number of accesses per page required to process a node of the R-tree is at most $2 \cdot (m+1)/2$. Similarly, the smallest sum of leaf depths is produced by a tree where all the leaves have the same depth. Thus, the number of accesses per page is at least $2 \cdot \log_2 m$.

The previous analysis assumes that only two orderings (one per coordinate) were used. Since a total of c orderings is used, the required number of accesses per page is between $2 \cdot (c-1) \cdot \log_2 m$ and $2 \cdot (c-1) \cdot (m+1)/2$. These bounds are often smaller for the root of the R-tree, where the number of clusters ranges between two and m.

Having analyzed the loading time for one R-tree node, we are ready to consider the entire R-tree. Including the cost due to sorting the average number of accesses per page for TGS is at least $3 \cdot c + 1 + 2 \cdot h \cdot (c-1) \cdot \log_2 m$ and at most $3 \cdot c + 1 + 2 \cdot h \cdot (c-1) \cdot (m+1)/2$, where h is the number of levels of the R-tree such that a node at that level is responsible for more data records than those that can fit in main memory. Clearly, h ranges between 1 and $\log_m n$. Given the amount of memory in typical database servers, h is often one, and external access is needed only while processing the root of the R-tree. Additional applications of the basic step can usually proceed entirely in main memory. Furthermore, in practice the number of accesses is much closer to the lower bound than to the upper bound. In this respect, the behavior of TGS is similar to that of the partitioning step of quicksort: the probability of repeatedly getting very small partitions (thus increasing the accesses to disk) is very small. For example, for a 2-dimensional set with m=100 and $h = 1$, the number of required disk accesses per page is around 21 for one ordering per dimension and 55 for two.

We point out that STR requires an average of 8 accesses per page. For Hilbert packing this average is only 4. In Section 5 we show that TGS results in trees that perform up to three times faster than those produced by other methods. Thus, we feel that the higher loading time of TGS is fully justified. However, TGS might be best suited for data sets where bulk loading need not be performed frequently.

Future work will concentrate on lowering the loading time while still producing high quality trees. One approach we plan to explore consists on modifying the basic split step so that it selects the best w cuts where $1 \leq w \leq m$ is specified by the user (at present $w = 1$). As w increases, we can expect that both the loading time and possibly the quality of the resulting tree will decrease. When $w = m$ TGS becomes similar to STR. Our work in the near future will try to characterize this tradeoff between loading time and resulting tree quality.

4 Experimental Methodology

In this section we discuss our experimental methodology. Our goal was to provide a sound implementation based comparison of the packing algorithms for numerous data sets. For each data set, R-trees are built using each of the three packing algorithms and then the respective R-trees are queried using the same set of randomly generated query points or query regions. For each test 10,000 searches are performed. No attempts at collecting confidence intervals have been made.

Since most real database systems utilize a buffer we include LRU buffering of the the R-tree and consider the performance for various buffer sizes. Our primary metric of interest is the average number of disk accesses (i.e., buffer misses) for the 10,000 searches. Since [6] shows that little is gained by pinning the top levels of the R-tree in the buffer, we simplify our study by using LRU for all nodes irrespective of their level in the tree.

We present results from base data which allows a comparison of the algorithms, but do little to explain why the observed performance occurs. To answer this harder question we synthetically modify the base data sets by changing aspects of the data sets one at a time. This allows us to observe how data set parameters affect observed performance in a more controlled setting. We first describe the base data sets, next the query generation, and finally how the data sets are synthetically altered.

4.1 Base Data Sets

In this subsection we describe our source data sets. They provide an initial comparison of the packing algorithms. For a unified framework, all data sets were normalized to the unit square.

All but one of our seed data sets (the uniform data set) can be characterized as having what we call from now on a *decreasing distribution* in areas and aspect ratios. We loosely define a *decreasing distribution* to be a distribution where larger values occur with lower probability than smaller values.

Our four base data sets are:

GIS: From the Geographic Information Systems, we work with the Long Beach data of the TIGER system of the U. S. Bureau of Census. This data set of 53,145 line segments and has been extensively used in past studies.

VLSI: A CIF data set of 453,994 rectangles provided by Bell Labs from the design of a chip [8]. Its highly skewed distribution in location, area, and aspect ratio makes this data set of particular interest.

Uniform: This is a synthetically generate set of 50,000 squares with a uniform distribution of areas and center point locations. Rectangle area is uniformly distributed between 0 and $2 \cdot a$, where $a = \frac{1}{50,000}$. The lower-left corners are uniformly distributed within the unit square. The upper right corner is chosen to give the desired area and it is truncated by the boundaries of the unit square when it does not fit.

Tile: This synthetic data set is created by dividing the unit square into 50,000 non-overlapping rectangular tiles. The size and aspect ratios of the tiles are determined as follows. Start with one tile, the unit square, and repeatedly split one of the tiles until the desired number of rectangles has been achieved. Tiles are selected at random, with a probability proportional to their area, and split across one dimension chosen at random with equal probability. The split line is chosen uniformly between the lower and upper coordinate values for the appropriate dimension. A sample data set containing 500 tiles is presented in Figure 5.

4.2 Query Generation

our experiments run a set of 10,000 point or region queries against each R-tree created. Point queries are generated as uniformly distributed points within the unit square. The lower left corner of a region query is uniformly distributed over the entire unit square. Region queries were set as squares with side dimensions 0.1 and 0.3 resulting in query regions of 1% and 9% of the unit square. Any region not fitting within the unit square was readjusted by truncation.

4.3 Modification of Base Data Sets

In order to gain insight into the cause of performance differences we modify the base data sets by varying the distribution of locations, areas, and/or aspect ratios. Each base data set is used as seed. Additional sets are constructed from the seed by varying one or more of the following geometry characteristics: data rectangle center point location (L or l), rectangle area (A or a), and/or rectangle aspect ratios (R or r). If we change the characteristic from skewed to uniform we designate the change with a capital letter appended to the seed name, for example seedname-L. If changed from uniform to skewed we use the lower case letter.

We define the aspect ratio to be the quotient of the smallest dimension (width or height) divided by the largest dimension. When generating set variants on the aspect ratio distribution, the longer edge is chosen with equal probability to be either the x-edge or the y-edge.

Consider for example a seed data set where the distribution of locations is not uniform. We redistribute the rectangles within the unit square by choosing a uniformly distributed center point for each rectangle. Thus, we would identify the new data set as seedname-L. Similar changes are made for seedname-R and seedname-A. We also combine changes such as uniformizing location and area, in which case we get seedname-AL. Consider again the Tile data set. Figures 6, 7 and 8 show the location, area, and aspect ratio variants for the Tile data set assuming 500 tiles.

Conversely, assume the rectangle locations in the seed are uniformly distributed. We create a new data set by modifying center locations to be non-uniform and also refer to this data set as seedname-l. Specifically, we try to create a distribution where objects are clustered with different densities in a small number of regions. To accomplish this, center locations are generated as follows. First generate a random number *numReg* (between 0 and 50) of rectangular regions with a uniform distribution of areas (such that the overall sum is 1). Then, for each region, a total of $(\lfloor 50,000/numReg \rfloor - 1)$ points is uniformly distributed inside the region. The total of 50,000 points is completed with uniformly distributed locations over the entire unit square. Note that each region has a different concentration of points per area. The resulting locations are shown in Figure 9.

If the seed data set areas are uniformly distributed, we modify them to the non-uniform variant seedname-a as follows. Associate a random variable, x_i, with each rectangle i. This random variable is assigned a uniformly distributed float between 0 and 1, and then squared. Let $V = \sum_i x_i$. Assume rectangle i currently has area a_i. Each rectangle now has its current area changed to $a_i \cdot \frac{x_i}{V}$.

If the seed data set aspect ratios are constant, as in the uniform seed data set, we modify them to non-constant by choosing each rectangle to have an aspect ratio uniformly distributed between 0 and 1.

5 Results

In this section we present results from our experimental studies for both point and region queries. Our primary comparison metric is the ratio of the average number of disk accesses needed for HS or STR relative to the average number of disk accesses for TGS. This allows direct comparison of the algorithms. In addition we provide the actual number of disk accesses for the TGS algorithm, thus allowing the reader to derive the number of disk accesses for the other algorithms if desired. The value used for the maximum number of entries per node, m, was set to 100. We examined buffer sizes of 10, 25, 50, 100, and 250 pages.

As noted in section 3, TGS can use any objective function when deciding where to partition the

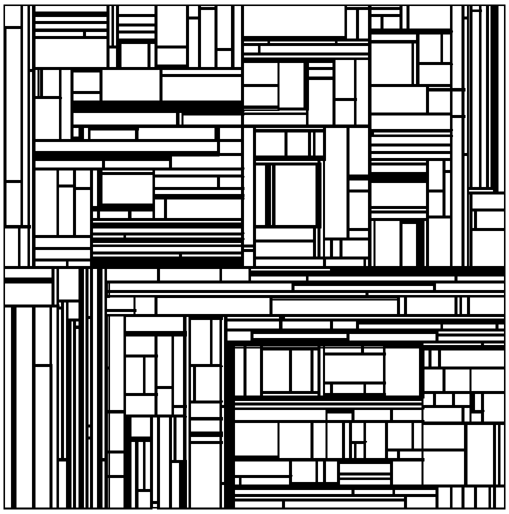


Figure 5: Tile data set of 500 rectangles

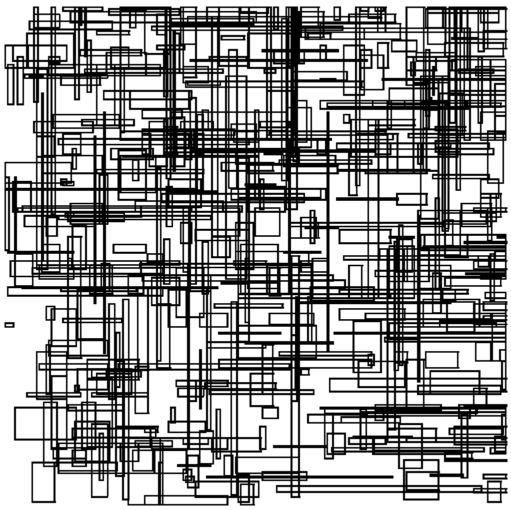


Figure 6: Tile of 500 rects., relocated (L)

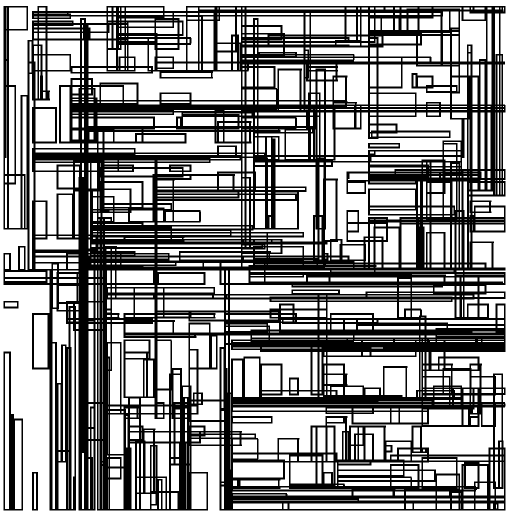


Figure 7: Tile of 500 rects., uniform area (A)

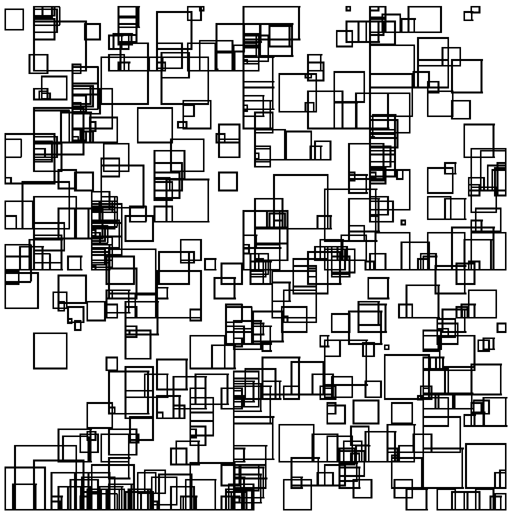


Figure 8: Tile of 500 rects., aspect ratio=1 (R)

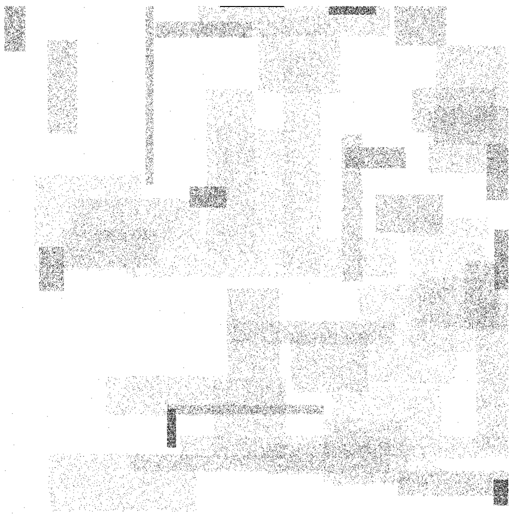


Figure 9: Regions of concentration of points (for relocation)

data set. We considered both just area and also a weighted function of area and perimeter [4] and observed no relevant difference. Thus, all results presented below assume TGS trees were constructed using just area as the objective function.

Also as noted in section 3, TGS can consider different orderings to construct the data partitions. For each data set, TGS was run three times using, in turn, the following orderings per dimension: (a) center coordinate, (b) minimum coordinate, and (c) minimum and maximum coordinate. We found that using the minimum coordinate resulted in queries requiring at least twice as many disk accesses as using min/max or center point.

Using only the center coordinate was competitive with min/max when the amount of skew was not significant. For example, using only the center coordinate resulted in performance within 5% of min/max for the uniform and GIS data sets. Thus, if it is known a priori that the data set is only mildly skewed the center point method can be used to reduce load time. For highly skewed data sets, such as the VLSI and Tile sets, the min/max method resulted in substantial query performance improvements relative to STR and HS. In all of the results presented below the min/max combination was used.

	performance ratio HS/TGS					performance ratio STR/TGS					TGS performance (DAs)				
data	10	25	50	100	250	10	25	50	100	250	10	25	50	100	250
seed	1.62	1.53	1.50	1.44	1.43	1.15	1.13	1.14	1.14	1.14	0.65	0.47	0.42	0.36	0.22
L	1.35	1.35	1.35	1.35	1.35	1.04	1.02	1.02	1.02	1.02	1.49	1.11	1.01	0.90	0.60
A	1.56	1.48	1.45	1.37	1.32	1.11	1.12	1.11	1.07	1.04	0.69	0.51	0.46	0.40	0.22
AL	1.35	1.33	1.33	1.34	1.34	1.05	1.04	1.03	1.03	1.03	1.52	1.13	1.04	0.92	0.61
R	1.61	1.51	1.50	1.42	1.35	1.12	1.07	1.08	1.09	1.05	0.58	0.42	0.38	0.33	0.20
RL	1.33	1.35	1.35	1.34	1.34	1.02	1.01	1.01	1.01	1.02	1.40	1.02	0.94	0.84	0.55
AR	1.63	1.54	1.45	1.39	1.35	1.11	1.07	1.05	1.03	1.00	0.56	0.41	0.37	0.32	0.20
ARL	1.34	1.34	1.35	1.36	1.34	1.01	1.00	1.01	1.01	0.98	1.38	1.02	0.94	0.84	0.55

Table 1: Point Query performance for GIS

Before presenting specific results we summarize two general trends. First, TGS performs better than STR or HS when there is significant skew in location, area, or aspect ratio. When two or more types of skew are present the differences is sometimes even more substantial then when only one type of skew is present. The data experiments are designed to synthetically reduce or increase skew. When skew is increased (resp. decreased), the difference increases (resp. decreases).

The second trend is that point queries result in a larger relative performance difference than region queries. There are two reasons for this. First, region queries return more of the data set and thus allow for a more naive search without significant degradation in performance [2]. Second, region queries implicitly have the effect of making the data appear more uniform. Thus, the TGS algorithm, which we will show performs best relative to the other algorithms when center point, areas and aspect ratios are more skewed, does not have as much skew to take advantage of. To understand why region queries have the effect of making the data appear more uniform consider the region query model of Kamel and Faloutsos [4]. It was shown that a region query is equivalent to a point query where all node MBRs are expanded in each dimension by an amount equal to the span of the query in the corresponding dimension. Thus, small MBRs (which are common in the seed data sets used in our studies) become significantly larger, decreasing the skew in terms of area, and decreasing the importance of skew in center point location.

5.1 GIS tiger data

The GIS data set has a decreasing distribution for both areas and aspect ratios and non-uniform distribution of positions. Therefore, we consider data set variants with uniform distributions of location (L), area (A), and/or a constant aspect ratio (R) of 1. For point queries, the number of disk

region query side = 0.1					
	10	25	50	100	250
HS/TGS	1.12	1.11	1.09	1.08	1.08
STR/TGS	1.05	1.05	1.05	1.05	1.04
TGS	10.83	10.18	9.31	8.18	5.25
region query side = 0.3					
	10	25	50	100	250
HS/TGS	1.04	1.04	1.04	1.04	1.03
STR/TGS	1.02	1.02	1.02	1.01	1.02
TGS	46.73	46.38	45.24	41.64	26.61

Table 2: Region Query performance for GIS

accesses for TGS, ratios of HS/TGS and STR/TGS are presented in Table 1. There is a column for each buffer size, and the first column specifies the data set variant used.

There is no significant difference in performance of TGS versus STR, but both are significantly better than HS. The variants show that when the data set becomes less skewed the HS/TGS and STR/TGS ratios become smaller. The TGS algorithm outperforms the other two when there is significant data skew (in terms of area, location, and aspect ratio). Thus, when the skew diminishes the difference of the algorithms diminishes.

The results for region queries are summarized in Table 2. Notice how TGS remains competitive against HS or STR, with an average improvement of 5% for small region queries. Also, as pointed out before, the larger the region query, the smaller the difference in performance, independent of which algorithms are being compared.

5.2 VLSI data

The areas and aspect ratios of this data set have a decreasing distribution. The locations are distributed according to the main components of the chip, thus, non-uniformly. For the data variants, the locations (L) and areas (A) are uniform, and the aspect ratios (R) are set to the constant 1. The total area of the data rectangles in the seed data set is 2.34, thus when the area variant (A) is created, each rectangle has its area generated uniformly between 0 and $2 * \frac{2.34}{453{,}994}$. For the aspect ratio variant (R), all rectangles retain their area and lines are converted to points.

The point query performance numbers are presented in table 3. The STR and HS algorithms require twice as many disk accesses as the TGS algorithm.

	Performance ratio HS/TGS					Performance ratio STR/TGS					TGS performance (DAs)				
data	10	25	50	100	250	10	25	50	100	250	10	25	50	100	250
seed	2.09	1.98	1.90	1.84	1.74	2.16	2.14	2.11	2.10	1.98	6.66	6.13	5.55	4.68	3.31
L	1.51	1.68	1.76	1.89	1.94	1.42	1.56	1.61	1.72	1.79	16.88	14.41	12.19	9.56	6.94
A	1.22	1.21	1.19	1.15	1.11	1.16	1.16	1.16	1.16	1.13	4.31	4.09	3.87	3.53	2.90
AL	0.99	1.00	1.00	0.98	0.97	0.87	0.89	0.89	0.89	0.88	6.69	6.37	6.10	5.65	4.81
R	1.22	1.21	1.17	1.08	1.00	1.05	1.04	1.03	0.97	0.91	3.23	2.91	2.53	2.09	1.54
RL	1.06	1.03	1.01	0.95	0.91	0.87	0.85	0.82	0.76	0.71	3.99	3.76	3.43	2.95	2.27
AR	1.38	1.39	1.41	1.39	1.36	1.14	1.15	1.15	1.13	1.11	1.96	1.78	1.55	1.27	0.98
ARL	1.37	1.36	1.36	1.36	1.36	1.06	1.04	1.03	0.99	0.98	2.30	2.17	1.96	1.66	1.32

Table 3: Point query performance for VLSI

region query side = 0.1					
	10	25	50	100	250
HS/TGS	0.99	0.99	0.99	0.99	0.96
STR/TGS	1.01	1.01	1.00	1.00	0.98
TGS	92.04	91.87	91.43	89.90	84.82
region query side = 0.3					
		25	50	100	250
HS/TGS	0.98	0.98	0.98	0.98	0.98
STR/TGS	0.99	0.99	0.99	0.99	0.99
TGS	401.5	401.4	401.1	400.0	394.1

Table 4: Region Query performance for VLSI

Each of the variants reduces data set skew, and in each case the STR/TGS and HS/TGS ratios become smaller. Note that the variants (A) and (R) have a larger impact than the variant (L), thus implying that area and aspect ratio play a more important role than location for this data set.

The results for region queries are summarized in Table 4. There is little difference in the performance of the three algorithms.

5.3 Uniform data

Point query performance results are presented in Table 5. The initial seed data set has no skew in terms of area, location, or aspect ratio. Thus, all three algorithms require a similar number of disk accesses. The variants *add* skew in terms of location (l) , area (a) , and aspect ratio (r). The location and aspect ratio variants increase the disk access ratios of STR/TGS and HS/TGS from unity to 1.07 - 1.59 depending on buffer size. This is because TGS can do a better job clustering the skewed data than STR and HS. Note that the area variant (a) makes little difference since the areas are still

	Performance ratio HS/TGS					Performance ratio STR/TGS					TGS performance (DAs)				
data	10	25	50	100	250	10	25	50	100	250	10	25	50	100	250
seed	1.32	1.32	1.33	1.33	1.34	0.99	0.98	0.99	0.99	1.02	1.56	1.18	1.10	0.97	0.62
l	1.55	1.48	1.43	1.40	1.34	1.20	1.17	1.12	1.08	1.07	1.08	0.81	0.72	0.59	0.29
a	1.33	1.33	1.33	1.33	1.32	1.01	0.99	1.00	1.00	1.00	1.57	1.18	1.10	0.97	0.61
al	1.48	1.44	1.43	1.39	1.29	1.18	1.17	1.16	1.11	1.06	1.22	0.87	0.76	0.62	0.31
r	1.59	1.55	1.52	1.48	1.38	1.27	1.26	1.23	1.21	1.11	2.01	1.57	1.43	1.27	0.79
rl	1.57	1.54	1.47	1.39	1.29	1.44	1.33	1.24	1.13	0.98	1.73	1.31	1.16	0.95	0.48
ar	1.55	1.51	1.48	1.43	1.34	1.25	1.23	1.21	1.17	1.08	2.06	1.62	1.47	1.30	0.81
arl	1.53	1.45	1.38	1.29	1.17	1.39	1.24	1.17	1.06	0.93	1.74	1.32	1.17	0.98	0.50

Table 5: Point query performance for UNIFORM

region query side = 0.1					
	10	25	50	100	250
HS/TGS	1.09	1.08	1.07	1.07	1.08
STR/TGS	0.98	0.97	0.96	0.96	0.97
TGS	12.31	10.76	9.84	8.49	5.14
region query side = 0.3					
	10	25	50	100	250
HS/TGS	1.04	1.04	1.04	1.03	1.03
STR/TGS	0.98	0.98	0.98	0.98	0.98
TGS	47.89	47.09	44.84	38.04	21.17

Table 6: Region Query performance for UNIFORM

so small they behave more like points. Also note that when two or more types of skew are added at once (ar, al, rl, arl) the ratio is increased more than if only one type of skew is added.

The results for region queries are summarized in Table 6. Once again there is little difference in the performance of the three algorithms.

5.4 Tile data

As shown if Figure 5, the base data set contains no overlapping, an almost uniform distribution of the locations, and typical decreasing distribution for both the areas and the aspect ratios. We consider the variants (R), (A), and (L). The point query results are present in table 7. For the base data set HS (STR) requires up to 3.03 (2.44) times more disk accesses that TGS.

The data variant experiments show that the significant improvements of TGS can be attributed to the skew in area, location, and aspect ratio. When skew is reduced by moving to variants (A), (R), and (L), the relative difference of STR/TGS or HS/TGS diminishes.

	Performance ratio HS/TGS					Performance ratio STR/TGS					TGS performance (DAs)				
data	10	25	50	100	250	10	25	50	100	250	10	25	50	100	250
seed	2.79	3.03	2.85	2.72	2.32	2.29	2.44	2.32	2.23	1.93	1.87	1.41	1.30	1.15	0.72
L	1.68	1.82	1.81	1.83	1.90	1.60	1.73	1.73	1.75	1.76	12.85	10.79	9.51	7.78	4.04
A	2.21	2.47	2.19	1.97	1.59	2.15	2.38	2.14	1.82	1.38	12.78	10.64	9.29	7.44	3.65
AL	2.51	2.84	2.80	2.67	2.49	2.38	2.68	2.67	2.56	2.34	15.61	13.47	11.92	9.78	4.97
R	1.33	1.33	1.33	1.32	1.31	1.01	1.01	1.01	1.01	1.00	1.58	1.19	1.10	0.97	0.61
RL	1.33	1.33	1.32	1.32	1.30	0.99	0.98	0.99	0.99	0.98	1.57	1.19	1.10	0.98	0.63
AR	1.33	1.33	1.33	1.33	1.30	1.01	1.00	1.00	1.00	0.98	1.53	1.15	1.07	0.95	0.60
ARL	1.32	1.30	1.31	1.30	1.29	0.99	0.97	0.98	0.98	0.98	1.55	1.17	1.09	0.97	0.61

Table 7: Point Query performance for TILE

	Performance ratio HS/TGS					Performance ratio STR/TGS					TGS performance (DAs)				
data	10	25	50	100	250	10	25	50	100	250	10	25	50	100	250
seed	1.21	1.23	1.22	1.21	1.21	1.12	1.13	1.12	1.11	1.11	16.40	14.75	13.39	11.40	6.63
L	1.18	1.19	1.20	1.19	1.18	1.14	1.15	1.17	1.16	1.14	36.01	35.13	31.89	26.88	15.12
A	1.32	1.33	1.35	1.21	1.04	1.29	1.30	1.31	1.18	0.98	39.21	38.44	34.86	29.14	15.72
AL	1.48	1.49	1.54	1.47	1.30	1.42	1.43	1.47	1.43	1.25	44.23	43.58	40.37	34.15	19.65
R	1.09	1.09	1.07	1.08	1.08	0.98	0.97	0.97	0.97	0.97	12.23	10.68	9.76	8.40	5.06
RL	1.09	1.08	1.07	1.07	1.08	0.97	0.97	0.96	0.96	0.96	12.31	10.77	9.85	8.49	5.13
AR	1.09	1.08	1.08	1.08	1.08	0.98	0.97	0.97	0.97	0.97	12.17	10.60	9.66	8.30	4.98
ARL	1.08	1.07	1.06	1.07	1.07	0.97	0.96	0.95	0.95	0.96	12.19	10.66	9.76	8.42	5.11

Table 8: $0.1x0.1$ Region Query performance for TILE

The results for query region are summarized in Tables 8, 9. The improvement of TGS relative to STR and HS remains for query regions of 1%, but disappears for query regions of 9%. We hypothesize that TGS remains better relative to STR and HS due to presence of extremely long rectangles (notice how once the aspect ratios are set to 1, the three algorithms present similar results).

6 Conclusions

We have presented Top-down Greedy Split (TGS), a new algorithm for the bulk loading of R-trees, that uses an adaptive preprocessing step by aggressively rearranging data objects while building the R-tree from top to bottom.

Two important factors allow TGS to outperform alternative approaches: (a) by proceeding top-down, our algorithm has more options to better partition the data at levels close to the root where efficient partitions are likely to have a bigger impact on performance, (b) by considering a larger collection of cuts (as compared to previous algorithms), TGS is better able to adapt to the geometric

	Performance ratio HS/TGS					Performance ratio STR/TGS					TGS performance (DAs)				
data	10	25	50	100	250	10	25	50	100	250	10	25	50	100	250
seed	1.06	1.07	1.07	1.07	1.08	1.02	1.03	1.03	1.03	1.03	56.40	55.59	53.22	45.54	24.57
L	1.05	1.05	1.06	1.07	1.07	1.03	1.03	1.03	1.05	1.05	87.84	87.48	85.14	75.62	40.81
A	1.13	1.13	1.14	1.16	1.03	1.12	1.11	1.11	1.12	1.00	90.31	90.00	88.00	77.20	40.28
AL	1.23	1.23	1.24	1.28	1.21	1.20	1.20	1.20	1.24	1.19	97.04	96.74	95.16	84.39	45.58
R	1.04	1.04	1.04	1.03	1.03	0.98	0.99	0.98	0.98	0.98	47.08	46.28	44.04	37.33	20.50
RL	1.03	1.03	1.03	1.03	1.03	0.98	0.98	0.98	0.98	0.98	47.83	47.04	44.78	38.06	21.26
AR	1.04	1.04	1.04	1.03	1.04	0.98	0.98	0.98	0.98	0.98	47.08	46.31	44.03	37.25	20.38
ARL	1.03	1.03	1.03	1.03	1.03	0.98	0.98	0.98	0.98	0.98	47.63	46.83	44.58	37.87	21.15

Table 9: $0.3x0.3$ Region Query performance for TILE

characteristics of the input data.

Previous approaches simplify the loading of region data by choosing a representative point, such as the center, for each input object and then constructing the tree based on the resulting point information. While TGS can clearly follow this same approach, it can also easily consider the "boundary" of the input objects, a factor that for certain types of data results significant improvements in performance.

We compared the performance of trees resulting from our algorithm with those built by other approaches through extensive experimentation on actual implementations. We found that R-trees constructed with TGS are up to three times more efficient than those built by STR and Hilbert, in terms of number of disk accesses. Specifically, TGS is clearly the method of choice for data sets with skew in location, area, or aspect ratio (where the larger number of cuts considered plays its role). Realistic data from various application domains allowed us to verify that this type of profile is common in practice.

Since TGS requires at least 2.6 times longer than competing approaches to bulk load large data sets (the overhead is mostly due to (b) above), it is best suited for cases where bulk loading need not be performed frequently (e.g. maps and other kinds of fairly static data).

At present, the user can trade loading time for query efficiency by reducing the number of orderings per coordinate. Future work will attempt to provide the user with finer control over this tradeoff by controlling not only the orderings considered but also the number of cuts found by one application of the basic split. Using the approach outlined at the end of Section 3 we hope to speed up loading without significantly sacrificing the quality of the resulting R-trees.

References

[1] Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B., "The $R^{\star}$-tree: an Efficient and robust Access Method for Points and Rectangles," Proc. ACM SIGMOD, p. 323-331, May 1990.

[2] Chazelle, B., "Filtering Search: A New Approach to Query Answering," SIAM J. Comput., vol. 15, p. 703-724, 1986.

[3] Guttman, A., "R-trees: a Dynamic Index Structure fo Spatial Searching," Proc. ACM SIGMOD, p. 47-57, 1984.

[4] Kamel, I., Faloutsos, C., "On Packing R-trees", Proc. 2nd International Conference on Information and Knowledge Management (CKIM-93), p. 490-499, Arlington, VA, November 1993.

[5] Kamel, I., Faloutsos, C., "Hilbert R-tree: An improved R-tree Using Fractals", Proc. International Conference on Very Large Databases 1995 (VLDB-95).

[6] Leutenegger, S.T., Lopez, M.A., "The Effect of Buffering on the Performance of R-Trees", University of Denver Technical Report number 96-2, submitted for publication.

[7] Leutenegger, S.T., Lopez, M.A., Edgington, J., "STR: A Simple and Efficient Algorithm for R-Tree Packing ", Proc. 13th International Conference on Data Engineering, p. 497-506, 1997.

[8] Lopez, M.A., Janardan, R., Sahni S., "Efficient Net Extraction for Restricted Orientation Designs", to appear in IEEE Transactions on CAD.

[9] A.L. Rosenberg, L. Snyder, "Time and Space Optimality in B-Trees," ACM Trasactions on Database Systems, vol. 6, no. 1, March 1981.

[10] Roussopoulos, N, Leifker, D., "Direct Spatial Search on Pictorial Databases Using Packed R-trees,", Proc. ACM SIGMOD, May 1985.

[11] Sellis, T., Roussopoulos, N., Faloutsos, C., "The R+ Tree: A Dynamic Index for Multidimensional Objects," Proc. 13th International Conference on Very Large Databases (VLDB-87), p. 507-518, September 1987.