On Optimal Node Splitting for R-trees *

Yván J. García R. 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

The problem of finding an optimal bipartition of a rectangle set has a direct impact on query performance of dynamic R-trees. During update operations, overflowed nodes need to be split (bipartitioned) with the goal of minimizing resultant expected query time. The previous algorithm for optimal node splitting requires exponential time. One contribution of this paper is a polynomial time algorithm for finding optimal bipartitions for any objective function whose value depends exclusively on the bounding hyper-rectangles of the ensuing partitions. The algorithm runs in $O(n^d)$ time where $d > 1$ is the number of dimensions of the input. Experimental studies indicate that the use of optimal splits alone results in improvements of query performance of only between 5% and 15% when compared to other heuristics. Thus, a second contribution is to demonstrate the near optimality of previous split heuristics, a fact that suggests that research should focus on global rather than local optimization issues. Finally, we propose a new dynamic R-tree insertion method that uses a more global restructuring heuristic when processing node overflows. When coupled with optimal splits our method results in improvements of up to 120% over the leading standard.

*This work has been partially supported by the National Science Foundation under grant IRI-9610240 and by the Colorado Advanced Software Institute under grants TT-96-05 and TT-97-06.

Proceedings of the 24th VLDB Conference
New York, USA, 1998

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 [Gut84] 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 [Gut84]. Thus, the first proposal for building an R-tree consists of simply inserting one record at a time until the input data is exhausted. Guttman's insertion algorithm operates by identifying a leaf to host the new record and updating its ancestors accordingly. An insertion to an already full node is handled by creating a new sibling to the full node and partitioning the rectangles of the full node plus the new record into two sets which are then stored in the new and previously full node, respectively. Note that insertion of the new sibling may require further splits along the path of ancestors of the previously full node. Since splits are the sole mechanism for creating nodes, these splits shape the resulting R-tree and affect its future query performance. Other dynamic approaches have been proposed [Bec90, Kam94, Sel87].

Guttman presented splitting algorithms with linear, quadratic, and exponential time complexity. He showed that the quadratic algorithm resulted in significantly better performance than the linear. Furthermore, he argued that the exponential algorithm was best, but due to its time complexity was not practical and hence only presented results for a limited number of small tests.

One of the main contributions of this paper is the development of a polynomial time splitting algorithm that is provably optimal, i.e. equivalent to Guttman's exponential algorithm, for a given objective function. Thus, we have reduced the complexity of the optimal splitting algorithm from exponential to $O(n^d)$, where $d > 1$ is the number of dimensions of the input data. In our experimental results we choose minimization of the expected number of node accesses as stated in [Kam93] (i.e., $A + q_x \cdot L_y + L_x \cdot q_y + q_x \cdot q_y \cdot N$, where A, L_x, L_y, q_x, q_y, N are the sum of areas, sum of node extents in x, y, extents of query in x,y and number of nodes respectively). Note that for point queries it is just the sum of areas.

The other major contribution of this work is a new R-tree insertion scheme which we call SIBLING-SHIFT (or simply SHIFT). We couple the optimal split algorithm with our SHIFT insertion method and show that resultant query performance can be significantly superior (up to a factor of 2.2) to that of Hilbert R-tree (the current best dynamic algorithm) even though it results in a slower insertion operation (on average by a factor of 2). Furthermore, we show that using splitting algorithms based on sweep-line over coordinates per axis or Hilbert values with our SHIFT framework does not work as well as when our optimal split algorithm is used.

The remainder of the paper is organized as follows. Background information on R-trees and descriptions of the algorithms considered are presented in Section 2. We introduce our proposed optimal splitting algorithm with its analysis in Section 3, and our new insertion method in Section 4. In Section 5 we describe our experimental methodology and present results for both real and synthetic data sets. We conclude in Section 6.

2 The R-tree and Its Algorithms

In this section we provide an overview of the R-tree and include the original insertion algorithms proposed by Guttman. We then briefly describe the Hilbert R-tree dynamic insertion algorithm. Readers familiar with this background material should skip to section 3.

2.1 The R-tree

An R-tree is a generalization of the B$^+$-tree developed for efficient processing of intersection queries on spatial databases. 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). The algorithms extend to higher dimensions in a straight forward manner, but to keep the exposition clear we only consider the 2-dimensional case.

An R-tree node is allowed to hold between m and 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 child pointer 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 satisfied 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.

An example of the graphical layout of the input rectangles and the MBR boundaries as well as the R-tree node structure is shown in Figure 1.

2.2 Guttman's General Insertion Algorithm

Guttman's general insertion algorithm proceeds as follows:

1. Choose the leaf L where to place new record R
2. if there is room in L for data item R insert it, otherwise split the entries in L with R into two subsets, one will become the entries of a newly created leaf L', and the other will remain as the only entries for L.
3. Propagate MBR changes upward from L (and the MBR of L' along with a pointer to L' if a split was done).
4. If the propagation resulted in a split of the root, create a new root whose children are the result of the old root split.

The leaf is chosen recursively descending from the root to the branch with the subtree whose MBR needs the smallest area enlargement to include the new record. Ties are solved by choosing the smallest resulting MBR area.

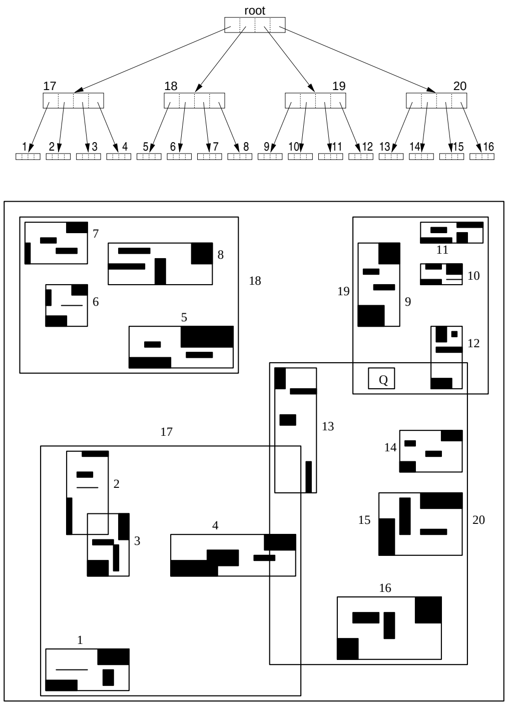

Figure 1: R-tree Example. Input data are filled rectangles. Node MBRs boundaries are a little bigger to differentiate them. Q is an example of a query region.

After insertion, changes are propagated to the root. First, the MBR in the parent's entry for L is updated. In case that there was an split, either the MBR of L' is inserted as a new entry in the parent if there is room, or an split is done with the parent node entries and the new MBR otherwise.

Guttman proposed three ways to split a node, and differentiate them by their time complexity which were linear, quadratic, and exponential. The split procedure was trying to solve the bipartition of a set of rectangles with an minimum value for the sum of the areas of the MBRs of the ensuing parts.

The three split algorithms are:

1. **Linear:** Chose two seeds, one for each resultant node, whose normalized distance in some dimension is greatest. For each remaining entry, assign it to the part that needs the least MBR area enlargement with ties being solved by the least MBR area, then the fewer entries, or any if no solution is found.

2. **Quadratic:** First, chose two seeds by considering all pairs of entries and determining the most wasteful if put together. The waste of grouping entries E_1 and E_2 is computed as the area of the MBR of E_1 and E_2 minus the areas of E_1 and E_2. The cost of assigning an entry to one node is the MBR area increase to include the entry in the node. To finish the split, assign the entry with highest difference of assignment costs until entries exhaustion. The node where to place the entry is decided as it is for Linear.

3. **Exponential:** (Exhaustive) Generate all possible groupings and chose the best.

2.3 The Hilbert R-tree

Proposed by Kamel and Faloutsos in [Kam94], this is a data structure based on the R-tree and an ordering of the leaves imposed by the Hilbert value. The Hilbert value is the length of the fractal space filling Hilbert curve from the origin up to the point the value is to be computed of. In brief the R-tree internal nodes are augmented to hold the largest Hilbert value of its descendant leaves. Insertion algorithms are guided by these Hilbert values. As a result, the leaves are sorted by Hilbert value. Another addition in the Hilbert R-tree, is the idea of creation of new nodes (splits) only when s neighboring sibling are full and a new record is to be inserted in one of them. If s neighboring siblings are not full overflow entries are evenly distributed among all s "cooperative" siblings. In our implementation we used the 2-to-3 split as it is purported to be the best compromise in time versus space utilization. According to [Kam94], the Hilbert R-tree is currently the best dynamic R-tree variant.

3 The Bipartition Algorithm

In this section, we begin by proving some basic results essential for the correctness of our algorithms. Then, we present the basic algorithm, analyze its complexity and modify it to derive a solution subject to the partitions having specific cardinalities. We simplify the presentation by discussing only the 2 dimensional case. Generalizations to higher dimensions are straightforward. For the rest of the paper, we use the notation in Table 1. The u-th defining value of a rectangle is simply the extreme value (i.e., maximum or minimum) of the u-th coordinate over all points contained in the rectangle.

3.1 Properties of Optimal Bipartitions

It is easy to see that the number of bipartitions of a set of n rectangles is exponential in n. However, many of the candidate bipartitons share the same pair of MBRs since the rectangles not touching the MBR boundary do not play a role in defining it. We show that for a cost function depending only on the MBRs the number of different MBR pairs is polynomial. Thus, in searching for an optimal bipartition, it suffices to consider only a polynomial number of them.

Table 1: Notation

S	set of input rectangles
n	cardinality of S
R_s	MBR of S
R_0, R_1	MBRs from a bipartition of S
C_u	rectangle's uth defining value
$C_{u,R}$	R's uth defining value
x	defining value for R's min. X
X	defining value for R's max. X
y	defining value for R's min. Y
Y	defining value for R's max. Y

Lemma 1 *The number of different MBRs from subsets of a set of n rectangles is at most n^4.*

Proof An MBR is defined by 4 values. Since there are n input rectangles, there are at most n different ways for selecting each of these values. □

Accordingly, since a bipartition requires two MBRs, the total number of MBR pairs to evaluate is at most n^8. We can now further reduce the number of candidate pairs.

Definition 1 *Considering the notation in Table 1, R_0 or R_1 is called an* **anchor MBR** *if it shares at least two of the defining values of R_s. The values thus shared with R_s are called* **anchoring defining** *values.*

Lemma 2 *Every pair R_0, R_1 of MBRs for a bipartition of S contains an anchor MBR and there are $O(n^2)$ different anchor MBRs.*

Proof Each of the four values defining R_s has to be present in either R_0 or R_1. Therefore, at least one of $\{R_0, R_1\}$ has to share at least two of the four R_s defining values. There are only $\binom{4}{2} = 6$ different ways for selecting a pair of R_s defining values. For a fixed pair of such values, only two other values are needed to complete the definition of an anchor MBR with only n choices for each value. Therefore, there are at most $6 \cdot n \cdot n$ different anchor MBRs. □

Since at least one MBR must be an anchor, the number of MBR pairs to evaluate is at most $O(n^2 \cdot n^4)$. This number is even lower for particular cost functions, as shown below.

Definition 2 *Let A and B be rectangles. A function $f(A, B)$ is said to be* **extent monotone** *if f increases monotonically with increases in either the x-extents or y-extents of either A or B. In other words $f(A, B)$ increases whenever one of $\{C_{X,A} - C_{x,A},\ C_{Y,A} - C_{y,A},\ C_{X,B} - C_{x,B},\ C_{Y,B} - C_{y,B}\}$ also increases.*

For example, $f(A, B) = \text{area}(A) + \text{area}(B)$ and $g(A, B) = \text{perimeter}(A) + \text{perimeter}(B)$ are both extent monotone functions.

Now, with the help of the following result, the number of MBR pairs to evaluate is only $O(n^2)$ when optimizing an extent monotone function.

Theorem 1 *Let $\text{cost}(R_0, R_1)$ be an extent monotone function. It suffices to consider $O(n^2)$ pairs when finding a bipartition that minimizes the value of $\text{cost}(\cdot, \cdot)$.*

Proof First we note that for a given MBR R_0, the optimal value for $cost(R_0, R_1)$ can be obtained from R_0 and a R_1 that is the MBR of those input rectangles not totally contained in R_0. This is obvious from the fact that any other R_1 will have a bigger extent and therefore, a higher $cost(R_0, R_1)$. Without loss of generality assume that R_0 is an anchor MBR. It follows that only $O(n^2)$ values of R_0 and hence $O(n^2)$ candidate MBR pairs need to be considered. □

We note that our claims, suitably modified, also hold whenever f is not extent monotone but $-f$ is.

3.2 The Basic Bipartition Algorithm

The basic algorithm operates by going thru each of the $O(n^2)$ pairs of MBRs and remembering the one with the best value for the given extent monotone cost function. Once the two MBRs that provide the best cost are found, then each input rectangle is assigned to its corresponding MBR. Rectangles contained in the intersection of the best MBR pair are assigned following a given criteria (for example, to the MBR that has the least number of rectangles assigned so far). The basic algorithm is presented in Figure 2.

We compute the MBR R_1 corresponding to an anchor MBR R_0 as the the MBR of the input rectangles that are outside (even partially) of R_0.

Definition 3 *For a given R_0, Let C_u be the u-th defining value of R_0. Let H be the open half-plane defined by C_u and not containing R_0. the* **MBR complementary to R_0 with respect to C_u** *is defined as the MBR of the input rectangles intersecting H.*

An example of the MBRs complementary wrt to all defining values of an MBR R_0 is shown is Figure 3. Clearly, R_1 is just the MBR of the set of MBRs complementary to R_0 with respect to each bounding value defining R_0 but not defining R_s.

The preprocessing step computes the complementary MBRs so that the computation of R_1 (wrt each anchor R_0) inside the triple loop of Figure 2 can be done in constant time. Since the outer loop performs $O(1)$ iterations the total time is $O(n^2)$ as desired.

When building the complementary MBRs, if the defining value C_u is C_X or C_Y, i.e., a maximum (resp.

```
{Preprocessing}
For each one u of {x,y,X,Y}
    Sort all the C_us
    Incrementally build the MBR complementary wrt C_u as the MBR
    of previous C_u in the ordering and said C'_us rectangle
    (where previous is subject to the way the ordering is traverse).

{Evaluate all relevant MBR pairs}
For each pair {C_U,C_V} of R_s defining values to consider (6 possible pairs)
    For each value of coordinate C_u ( u differ from U and V )
        For each value of coordinate C_v ( where v differs from U,V, and u )
            Let R_0 be the rectangle {C_U,C_V,C_u,C_v}
            If R_0 is a proper MBR (i.e.  each border is touched
            by at least one interior input rectangle) then
                Let R_u be the MBR complementary to R_0 wrt C_u
                Let R_v be the MBR complementary to R_0 wrt C_v
                Let R_1 be the MBR of R_u and R_v
                Remember R_0,R_1 if cost(R_0,R_1) is better than the optimal cost so far

{classify input rectangles by optimal MBR pair}
For each input rectangle, assign it to the MBR that totally contains
it or to any in case of ambiguity.
```

Figure 2: The basic optimal bipartition algorithm

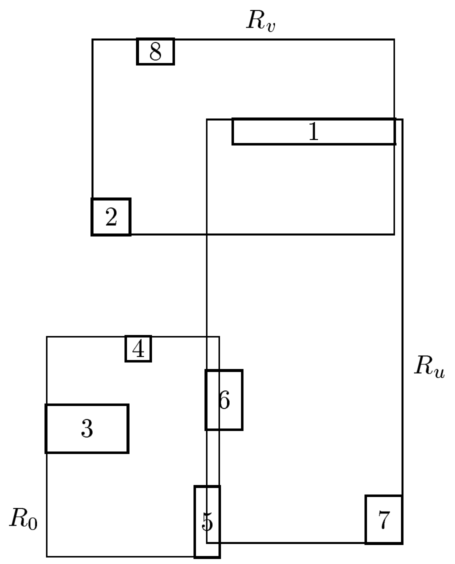

Figure 3: R_u is the MBR complementary to R_0 wrt C_{X,R_0}, and R_v is so wrt C_{Y,R_0}. The values C_{x,R_0} and C_{y,R_0} are not considered since R_0 shares those values with R_s.

C_x or C_y, i.e., a minimum) then the input list is traversed in descending (resp. ascending) order. Notice that the sorting step takes $O(n \log n)$ time and the traversal is linear, so, the quadratic complexity of the whole bipartition algorithm holds.

A simple generalization of this algorithm solves the optimal bipartition problem for any fixed $d > 1$ number of dimensions with running time of $O(n^d)$. The algorithm for d dimensions will iterate over d-tuples of rectangle defining values for each d-tuple of anchoring defining values.

3.3 Imposing Cardinality Constraints

The previous (unconstrained) algorithm may result in partitions with very different cardinalities. Node splitting often requires partitions with roughly the same number of rectangles. We refer to this as a bipartition with cardinality constraint, where R_0 (resp. R_1) must contain a number of rectangles in a user specified range. A simple brute force approach that runs in $O(n^3)$ time would compute the cardinality of both R_0 and R_1 with each iteration of the innermost loop of Figure 2. However, a few changes to the basic algorithm are sufficient to compute bipartitions with cardinality constraints while preserving the quadratic run-

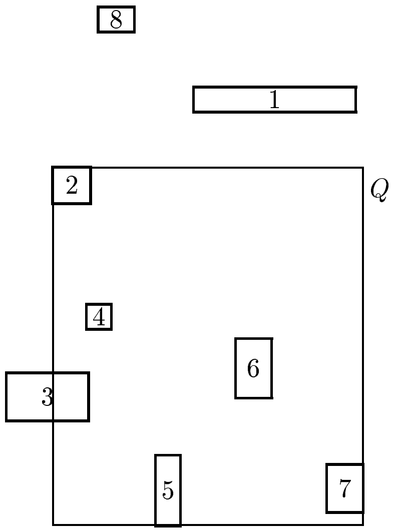

Figure 4: For corner $C_{x,2}, C_{Y,2}$ of rectangle 2 (upper left corner), Q is the MBR containing rectangles 2, 5 and 7. The corner cardinality is 5. Note that the x, Y corner of rectangle 3 is not counted as it is outside Q.

ning time.

The first change we will need is to check that the range of cardinalities of R_0 and R_1 are valid before evaluating their cost. We specify *range* since rectangles in $R_0 \cap R_1$ can be assigned to either of R_0 or R_1. Notice that said test requires the cardinality of the intersection which is easily computed as the sum of both cardinalities minus n. From now on we consider only the maximum cardinality of R_0 and R_1. To preserve the quadratic complexity, the cardinality test uses precomputed information which we proceed to define:

Definition 4 *Let R be an MBR and $(C_{u,R}, C_{v,R})$ one of the corners of R. Let Q be the rectangle defined by said corner and a corner of R_s such that $R \cap Q = R$. The* **corner cardinality** *of Q, denoted* $\mathrm{Card}[C_{u,R}, C_{v,R}]$*, is the number of input rectangle corners of the same orientation (e.g. top left) contained in Q.*

Figure 4 illustrates how to compute Q and corner cardinality for the upper left corner of rectangle $R = 2$.

Consider now the problem of computing the cardinality of R. We consider 5 cases depending on the number of defining values of R_s shared by R (see Figure 5):

Four values. That means that $R = R_s$. Clearly, the cardinality of R is n and its pairing MBR has cardinality 0.

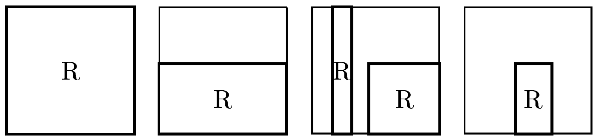

Figure 5: MBR R with 4,3,2, and 1 R_s defining values.

Three values. The cardinality of R is that of any of its corners which is not a corner of R_s.

Two values. We call a corner interior if it is strictly inside R_s (i.e., not on the boundary of R_s). If R has an interior corner, its cardinality is that of said corner. Otherwise, the cardinality of R is the sum of the cardinalities of any two (non-interior) corners of R in the same R_s edge minus n.

One value. Clearly R has two interior corners. Let P be a corner obtained as the point of intersection between the line through the two interior corners of R and the boundary of R_s. The cardinality of R is simply the sum of the cardinality of its two interior corners minus the cardinality of P.

No value. This case can be generated by the algorithm only if R's pairing MBR is a four-values case containing all n input rectangles (Note that R is not an anchor MBR). So, the cardinality for this case is 0.

Assuming that $\mathit{Card}[C_u, C_v]$ is available in constant time, the cardinalities of a pair R_0, R_1 can be obtained without affecting the time complexity of the basic bipartition algorithm (given in Figure 2) as long as any additional preprocessing does not exceed such running time.

The addition to the preprocessing step that allows to have the cardinality of the corners available in constant time is presented in Figure 6 (to be done just after the original preprocessing). Again, *previous*() is subject to the order of traversal. The ordering is traversed in such a way that the first value is the closest to the global corner diagonally opposed to the one being filled up (for example, if the process is creating $\mathit{Card}[C_x, C_y]$, then the first values correspond to the max C_x and max C_y). It suffices to have a series of linearly built cross references in order to know to which rectangle a given coordinate belongs to.

The last addition to the preprocessing step imposes a quadratic complexity due to the double for loop of n elements each. That implies that the overall complexity for the bipartition algorithm still holds.

One final modification to the basic algorithm concerns the classification of the input rectangles. The

```
For each pair u,v of possible corners indices (4 pairs)
    For each C_u in the proper order
        Let count be zero
        For each C_v in the proper order
            If C_u and C_v belong to the same rectangle then increment count
            Assign count to Card[C_u, C_v]
            If there is a previous(C_u) then
                Increment Card[C_u, C_v] by Card[previous(C_u), C_v]
                If there is a previous(C_v) then
                    decrement Card[previous(C_u), previous(C_v)]
```

Figure 6: Computing corners cardinality

rectangles that are not in the intersection of the resulting MBRs should be done first. Each of these rectangles goes to the MBR that contains it. Then, the assignment of the rest of the input rectangles (the ones in the intersection of R_0 and R_1) should be done one at a time to either the MBR that contains the least number of assignments when possible or to any otherwise.

4 A High Utilization Insertion Algorithm

In this section, we describe our insertion method for a dynamic R-tree. Then, we briefly explain the other two splitting algorithms aside from the optimal bipartition used in combination with our insertion method.

Two orthogonal splitting issues impact search performance: (1) minimization of some objective function (such as area or perimeter) of the split algorithm and (2) node utilization.

Previous work has shown that improving space utilization, i.e., maintaining a higher occupancy per node, can improve the resultant search performance. Thus, even if a split algorithm provides optimal splits with regards to the objective function, further improvements can be made by maintaining high node utilization. The Hilbert R-tree [Kam94] achieves improved node utilization by using 2-3 splitting.

In addition to the split algorithm (local optimization), overall tree structure (global optimization) has a significant impact on search performance. The superior performance of packing algorithms [Rou85, Kam93, Leu97] is attributable to both better node occupancy and better tree structure. The dynamic Hilbert R-tree not only achieves good overall node utilization, but also improves overall tree structure by following the Hilbert order.

We have devised a new dynamic algorithm incorporating improved node utilization and improved tree restructuring. Our new insertion algorithm, called SHIFT, only creates a new node if no sibling can be found to absorb one of the subsets created by a split. In addition, the SHIFT algorithm improves overall structure by heuristics designed to reorganize entries among siblings.

The algorithm selects the insertion path by greedily choosing at each node the branch that optimizes a given cost function. (The objective function used in our experiments is the expected number of node accesses as described in [Kam93].) Furthermore, our algorithm differs from other solutions in the way that node overflows are handled. If a node is full, it is split into two sets. One set remains in the node and the other set, say set B, is inserted into one of the node's siblings. The sibling is chosen so as to minimize the given objective function. If the chosen sibling can not accommodate the entire set B the sibling is likewise split. This continues until either a sibling that can accommodate the offered rectangle set is found or all siblings have been tried and failed. In the later case, a new node is created and an entry put in the parent node. This insertion into the parent node is handled recursively.

We now describe in detail our approach for inserting into a full node. For simplicity, we use the same symbol to denote both a node as well as the set of rectangles stored in that node. Let E be a node into which we wish to insert a set of rectangles R. Initially R is a single rectangle, but, as discussed below, subsequent iterations may require the insertion of more than one rectangle. Let P be the parent of E (we discuss the root node later). Initially, flag E as dirty and all its siblings as clean. There are two cases to consider depending on whether or not E has enough room for R. If E has no more than $M-|R|$ rectangles, add R to E and stop. Otherwise, split the rectangle set $E \cup R$ using a splitting algorithm (e.g., optimal bipartition). This results in two rectangle sets, E_a and E_b, one set to remain in node E, the other to be inserted into a sibling node. Among siblings of E, find the clean node F_a (resp. F_b) whose cost increases the least when including E_a (resp. E_b). Without loss of generality, assume that the cost of F_b increases less than that of F_a, i.e., $\text{cost}(\text{MBR}(F_a \cup E_a)) - \text{cost}(F_a) > \text{cost}(\text{MBR}(F_b \cup E_b)) - \text{cost}(F_b)$. Mark F_b's entry as dirty. To complete this iteration and proceed with the next one, the entry for E is updated to be associated with E_a, reset E to be the entry for F_b and reset R to E_b. Now repeat the splitting of the new E by inserting the new R into E. Continue until a node that can accommodate R is found, or all siblings have been tried and failed.

Note that a true split (i.e. an increase in the number of entries of P) occurs only when no clean sibling of E is found to absorb R, in which case a new node is created for R. Also note that an overflow of the root node always results in a split since the root has no siblings.

In the worst case, an insert into a full node could require $M-1$ disk accesses (one for each sibling). However, our experimental results in Section 5 show that SHIFT insertions are on average 2 times slower than Hilbert insertions and never more than 2.86 times slower. This is because few insertions require splitting of the node. When a split is required it may require up to $M-1$ disk accesses, but this cost is amortized across all insertions.

Note that our method for dealing with overflow is orthogonal to the actual split algorithm used. Thus, we consider the following three combinations of SHIFT and splitting algorithm:

- SHIFT-COORD: The splitting is done by passing a sweep-line by each axis, and remembering the rectangle center value that provides the minimum sum of cost for the MBRs of the partitions left at each side of said value.
- SHIFT-HILBERT: Same as before, but Hilbert values are used instead of rectangle defining values.
- SHIFT-OPTIMAL: Using the optimal split.

5 Experimental Results

In this section we present our experimental methodology and the obtained results. Since Hilbert R-trees are the current state of the art in low dimensional R-trees, our main focus is on comparing four algorithms: Hilbert R-trees (HILB), SHIFT-OPTIMAL, SHIFT-HILBERT and SHIFT-COORD. In addition, we evaluate the performance of Guttman's insertion algorithm [Gut84] under two different splitting policies: Guttman's quadratic splitting (GUT) and optimal splitting (OPT).

5.1 Methodology

We implemented all R-tree algorithms, assuming the maximum number of M entries per node to be 100, and present results from both synthetic and real data. All data sets were normalized to fit within the unit square for ease of study. We consider the following data sets:

- **TIGER:** This is the Long Beach data set coming from the TIGER system of the U. S. Bureau of Census. This data set of 53,145 rectangles has been extensively used in past studies.
- **VLSI:** A CIF data set of 97,069 rectangles provided by Bell Labs from the design of a chip [Lop96]. Its highly skewed distribution makes this data set of particular interest.
- **UNIF:** This is a synthetically generated 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.
- **CLUSTER:** This synthetic data set is created by fixing rectangle locations, areas and aspect ratios (quotient of the smaller extent over the greatest) so as to make it irregular. The rectangle centers are generated beginning with a random number *numReg* (between 0 and 50) of rectangular regions with uniform distribution of areas (such that the overall sum is 1). Then, for each region, $(\lfloor 50,000/numReg \rfloor - 1)$ points are uniformly distributed inside the region. A total of 50,000 points is completed with uniformly distributed locations over the entire unit square. A sample distribution is shown in Figure 7. Areas and aspect ratios are generated by uniform distributions. An additional uniformly distributed random variable was used to determine with equal probability whether the rectangle is wide or tall.

As shown in [Leu98], the number of disk accesses given a specific buffer size is a more meaningful performance metric than number of nodes accessed. Thus, we implemented an LRU buffer used in conjunction

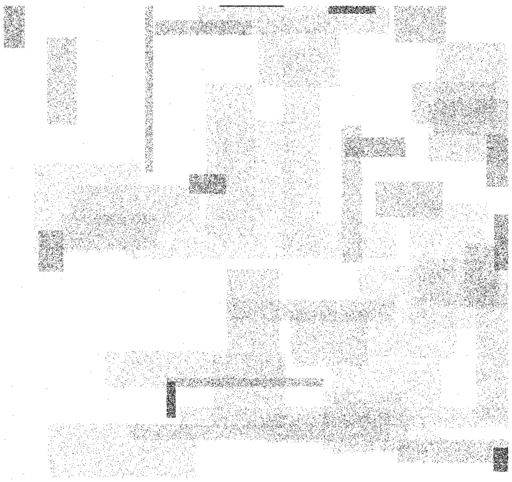

Figure 7: Regions of concentration of points

with a raw disk partition, to prevent false buffering by the OS, and report the number of disk accesses for various buffer sizes.

Specifically, we consider buffer sizes of 10, 25, 50, and 100 pages. For the VLSI data set, this represents roughly 1%, 2.5%, 5% and 10% of the final R-tree. For the other data sets, the corresponding percentages are 2%, 5%, 10% and 20%. Each presented data point is obtained from averaging 10,000 point queries uniformly distributed within the unit square.

5.2 Results

We first consider a comparison without the SHIFT heuristic. We compare Hilbert R-trees (HILB), Guttman's quadratic (GUT), and Guttman's insertion algorithm using optimal splitting (OPT). Different values of node utilization affect overall R-tree performance as previously shown in [Bec90]. To address this we consider minimum node occupancies of 0,10,20,40, and 50% of M. Therefore, we used the full version of the optimal bipartition algorithm with cardinality constraints.

In Table 2 we present results for the tiger longbeach data set. There is a column for each buffer size and a row for each method. For the OPT algorithm there is also a line for each node utilization considered.

First, compare the number of disk access of the quadratic algorithm (GUT) with the optimal split algorithm with node utilization of 50% (OPT 0.5). In general there is very little difference (less than 15%) in resultant query performance. Thus, there is little gain from optimal splitting. This is because the initial splits have a profound impact on final Rtree structure, and a specific split only considers one node, not the whole tree. A split that is optimal for the node in question may actually cause global structure to be worse than a suboptimal split.

Table 2: Disk accesses for tiger data

Method	m/M	10	25	50	100
HILB		1.0958	0.7572	0.6355	0.5390
GUT		1.1921	0.8754	0.6555	0.5464
OPT	0.50	1.0939	0.7710	0.6009	0.5306
	0.40	1.0131	0.7161	0.5652	0.4934
	0.20	0.9259	0.6467	0.5255	0.4598
	0.10	0.9280	0.6555	0.5316	0.4667
	0.00	1.0277	0.7191	0.5481	0.4749

These results indicate that optimal splitting of a node, a local optimization problem, does not necessarily improve global tree structure. Thus, improving overall tree structure is better done by other methods. Possible methods include increasing the amount of freedom during splitting by allowing under-filled nodes [Bec90], using packing algorithms [Rou85, Kam93, Leu97], or taking into consideration more global properties of the tree such as the forced reinserts of the R* tree [Bec90] and the Hilbert ordering structure of the Hilbert tree [Kam94], or the restructuring of our new SHIFT algorithm.

To improve tree structure we first consider allowing under-filled nodes, thus allowing more split choices to be considered. Table 2 shows that query performance can be improved by 10-45%, but that the determining the "optimal" minimum utilization is not straight forward. Note, Beckmann *et al* suggest a minimum utilization of 40% works well in practice, but for the tiger data we have found a minimum of 10-20% works best. Thus, allowing more freedom in split choices, at the expense of worse node utilization can actually improve search performance.

We now consider the opposite approach: increasing node utilization by using our SHIFT insertion framework. As a reminder, this policy not only improves node utilization, but also restructures sibling nodes by trying to find a new home for the newly split portion of the node.

We first consider the relative insertion speed of the SHIFT family versus Hilbert R-trees. In Table 3 there is a row for each data set used in our experiments. The first and second columns are for the total number of node accesses performed during the construction of the tree for SHIFT and HILB respectively. The last column, is the node access ratio of SHIFT over HILB. For SHIFT, we took the worst case out of the three combinations of splitting algorithms (SHIFT-OPT, SHIFT-HILB, SHIFT-COORD). SHIFT is never more than 2.86 times slower than HILB. Below we show that this slower insertion speed is justified by faster search speeds.

Table 3: Relative insertion speed

Data set	Shift	Hilb	Shift/Hilb
Tiger	258,579	161,934	1.5968
VLSI	555,664	303,175	1.8928
Unif	434,539	152,263	2.8538
Cluster	300,323	151,738	1.9792

Since SHIFT is geared towards better node utilization, we compare the number of nodes in the R-trees built by SHIFT, HILB and a packing algorithm (MIN) which results in the minimum number of nodes needed for the tree. Those numbers corresponds to columns 1, 2, and 3 respectively in Table 4. Columns 4 and 5 are the ratios (column 3 / column 1) and (column 3 / column 2) respectively. The SHIFT framework achieves a better utilization than HILB.

Table 4: Node utilization

data set	Shift	Hilb	Min	Min/Shift	Min/Hilb
Tiger	580	628	539	0.9293	0.8582
VLSI	1043	1137	982	0.9415	0.8637
Unif	555	572	506	0.9117	0.8846
Cluster	545	579	506	0.9284	0.8739

Finally, we present the relative query improvement of SHIFT-OPTIMAL over SHIFT-HILBERT, SHIFT-COORD, and HILB as the ratio average of number of disk accesses of each algorithm over SHIFT-OPTIMAL. The first rows are the ratio of each competing algorithm. The last row is the actual number of disk accesses for SHIFT-OPTIMAL from which actual performance of each algorithm can be derived. We experimented with square queries with sides of length 0.0 (point query), 0.01, 0.1, and 0.3 generated in a manner similar to that for the UNIF data set. There is a column for each query side length. Choice of buffer size did not affect the relative performance so we only consider one buffer size for each experiment.

Table 5: Improvement for TIGER (buffer size=10)

	Disk accesses ratio			
algorithm	0.0	0.01	0.1	0.3
HILB	1.34	1.26	1.12	1.10
SHIFT-COORD	0.85	0.93	1.07	1.13
SHIFT-HILBERT	1.54	1.40	1.13	1.06
	Actual disk accesses			
SHIFT-OPTIMAL	0.82	1.33	10.95	54.54

Consider first the TIGER data set results. In Table 5 we see that SHIFT-COORD works best for small queries and SHIFT-OPTIMAL works best for larger queries. SHIFT-OPTIMAL requires 10% - 34% fewer disk accesses than the Hilbert R-tree.

Next consider results for the VLSI data set. In Table 6 we see that SHIFT-OPTIMAL is best overall and requires up to 2.2 times fewer disk accesses than Hilbert.

Table 6: Improvement for VLSI (buffer size=10)

	Disk accesses ratio			
algorithm	0.0	0.01	0.1	0.3
HILB	2.21	2.02	1.34	1.16
SHIFT-COORD	1.25	1.30	1.25	1.18
	Actual disk accesses			
SHIFT-OPTIMAL	3.16	4.24	21.36	87.24

Table 7: Improvement for UNIF (buffer size=100)

	Disk accesses ratio			
algorithm	0.0	0.01	0.1	0.3
HILB	0.98	1.00	1.02	1.02
SHIFT-COORD	0.99	1.05	1.15	1.17
SHIFT-HILBERT	1.28	1.27	1.16	1.09
	Actual disk accesses			
SHIFT-OPTIMAL	1.34	1.87	10.14	44.61

Next consider the UNIF data set. In Table 7 we see that SHIFT-OPTIMAL is slightly better than the other methods but there is not a significant difference. We surmise that the uniformity of the data set makes it easy for all of the methods to structure the tree effectively.

Finally, consider the results in Table 8 for the CLUSTER data set. SHIFT-OPTIMAL requires 8% - 45% fewer disk accesses than Hilbert, but for point queries SHIFT-COORD is marginally better than SHIFT-OPIMAL.

In general, SHIFT produces trees requiring significantly fewer disk accesses than Hilbert. Furthermore, SHIFT-COORD works better than SHIFT-OPTIMAL for small queries and the reverse is true for large queries.

Table 8: Improvement for CLUSTER (buffer size=25)

	Disk accesses ratio			
algorithm	0.0	0.01	0.1	0.3
HILB	1.45	1.33	1.15	1.08
SHIFT-COORD	0.89	1.00	1.13	1.15
SHIFT-HILBERT	1.54	1.46	1.27	1.18
	Actual disk accesses			
SHIFT-OPTIMAL	0.91	1.46	10.68	52.01

6 Contributions and Conclusions

In this paper we have made the following contributions:

- We have reduced the time complexity of optimal bipartition of a set of 2 dimensional rectangles (node splitting) from exponential to $O(n^2)$. The optimality is measured by a fairly general user-defined objective function.
- We have generalized the $O(n^2)$ optimal bipartitioning algorithm to handle constraints of partition cardinality range (node utilization).
- We have generalized the optimal bipartitioning algorithm for a fixed number $d > 1$ of dimensions for which the complexity is $O(n^d)$.
- We have provided empirical results which show that optimal splitting alone results in R-trees whose query performance is up to 15% better than Guttman's quadratic algorithm.
- We have devised a new insertion framework called SHIFT and show that when SHIFT is combined with optimal splitting resultant trees provide up to 120% improvement relative to Hilbert R-trees.
- We have also shown that SHIFT works best when coupled with optimal bipartition instead of other splitting algorithms.

References

[Bec90] 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.

[Gut84] Guttman, A., "R-trees: a Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOD*, p. 47-57, 1984.

[Kam93] Kamel, I., Faloutsos, C., "On Packing R-trees", *Proc. 2nd International Conference on Information and Knowledge Management*, p. 490-499, Arlington, VA, November 1993 (CKIM-93).

[Kam94] Kamel, I., Faloutsos, C., "Hilbert R-tree: An improved R-tree Using Fractals", *Proc. International Conference on Very Large Databases*, 1994 (VLDB-94).

[Leu98] Leutenegger, S.T., Lopez, M.A., "The Effect of Buffering on the Performance of R-Trees", *Proc. 14th International Conference on Data Engineering*, 1997 (ICDE-97).

[Leu97] 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 (ICDE-97).

[Lop96] Lopez, M.A., Janardan, R., Sahni S., "Efficient Net Extraction for Restricted Orientation Designs", *IEEE Transactions on Computer-Aided Design, vol. 15, no. 9*, p. 1151-1159, September 1996.

[Rou85] Roussopoulos, N, Leifker, D., "Direct Spatial Search on Pictorial Databases Using Packed R-trees,", *Proc. ACM SIGMOD*, May 1985.

[Sel87] Sellis, T., Roussopoulos, N., Faloutsos, C., "The R+ Tree: A Dynamic Index for Multidimensional Objects," *Proc. 13th International Conference on Very Large Databases*, p. 507-518, September 1987 (VLDB-87).