FINAL REPORT

Colorado Advanced Software Institute

Efficient Access Methods for Multidimensional Data

Principal Investigators:	Scott T. Leutenegger, PhD Associate Professor Department of Mathematics and Computer Science University of Denver Mario A. Lopez, PhD Associate Professor Department of Mathematics and Computer Science University of Denver
Graduate Student:	Yvan Garcia, MS Department of Mathematics and Computer Science University of Denver
Collaborating Company:	MetaComp Inc. J.C. Franchitti, President

Project Title:	Efficient Access Methods for Multidimensional Data
Principal Investigators:	Scott T. Leutenegger, PhD
	Mario A. Lopez, PhD
University:	University of Denver
Collaborating Company:	MetaComp Inc.
Company Representative	J.C. Franchitti, President

As an authorized representative of the collaborating company, I have reviewed this report and approve it for release to the Colorado Advanced Software Institute.

Signature

Date

Abstract

Interest in multidimensional data has been steadily rising in the database community. Storage and subset retrieval of multi-dimensional data is necessary for scientific, geographic, temporal and business data. Our research has concentrated on developing techniques for efficient loading, retrieval, and update of multi-dimensional data. Current commercial systems (with the exception of one) do not support multi-dimensional efficiently. Instead, they rely on traditional one-dimensional techniques, resulting in sub-optimal access methods with poor performance for large databases. The one system that does provide multi-dimensional indexing only two and three dimensional indexing using simpler and less efficient algorithms than those developed in this work. Our prototype code has been handed over to MetaComp for integration into their intranet toolkit.

1 Background and Problem Description

Interest in multidimensional data has been steadily rising in the database community. Storage and retrieval of multidimensional data is necessary for many business, scientific, geographic, and engineering applications. Traditional secondary storage techniques such as B-trees and hash tables are not suitable for dealing with multidimensional data: by concentrating on one-dimension at a time, large quantities of extraneous data may be retrieved. A number of techniques designed specifically for multidimensional data have been proposed [1, 4, 5, 9, 10, 12].

In the area of spatial databases industry lags behind as only one commercial system (Informix) incorporates true multidimensional indexing. Even that product only provides a small subset of the methods we have developed in the last two years. Several third party software products complement relational On-Line Transaction Processing Systems with multidimensional On-Line Analytical Processing (OLAP) capabilities. Since commercial relational systems do not support efficient multidimensional indexing, OLAP products create and maintain analytical databases which duplicate needed data from the transaction databases. These OLAP capabilities are restricted to business data, and require costly data replication algorithms due to the lack of multidimensional indexing support. The Illustra product implements only a subset of the efficient multidimensional database indexing and spatial data support capabilities that the PIs intend to focus on. We have developed efficient techniques for multi-dimensional indexing (as outlined in section 2) and delivered our prototype to MetaComp for incorporation into their Intranet Toolkit.

An example application would be a business data base. Finding all employees between 30 and 35 years of age and earning more than $60,000 is an example of two dimensional region query. Adding additional dimensions such as zip code range would further reduce the number of qualifying records.

XXX put in figure here.

2 Objectives

Applications where the underlying data does not change are said to be static, and arise naturally in many areas, such as scientific databases, GIS, and VLSI CAD. In contrast, data sets that are subject to frequent updates are said to be dynamic. Our research concentrated on developing techniques for efficient loading, retrieval, and update of both static and dynamic multidimensional data. Our algorithms are based on R-trees (described in sect. XXX).

Primary research goals:

- **Static Data:** Develop fast bulk-loading algorithms that also result in high quality R-trees, as measured by node utilization and average query time.
- **Dynamic Data:** Develop new insertion and deletion algorithms that guarantee high quality R-trees, relative to the R-tree obtained by applying the bulk-loading algorithm to a static data set.
- **Parallel R-trees:** The original proposal stated we would conduct research in parallel R-trees. MetaComp Inc decided the parallel R-tree research was no longer of interest to them and hence is not included in this report. Note in our publications section that we did conduct research in the area but do not include it in this report.

Primary technology transfer goal:

- Aid integration of the new bulk loading and dynamic R-tree algorithms into the extensible hybrid DBMS being developed by MetaComp Inc.

3 Approach

R-trees [4] have been shown to be one of the most promising approaches for solving region and point queries efficiently. An R-tree is a hierarchical data structure derived from the B-tree to perform rectangular intersection queries on a collection of rectangles which can change in time. Other geometric objects can be handled by storing their bounding boxes. Each node of the R-tree stores a maximum of k entries. Each entry consists of a rectangle R and a pointer P. At the leaf level, R is the bounding box of an actual object pointed to by P. At internal nodes, R is the bounding box of all rectangles stored in the subtree pointed to by P. Figure 1 illustrates an R-tree with 3 levels (level 0 is the root) assuming that a maximum of $k = 4$ rectangles fit per node. There are 64 rectangles represented by the small dark boxes. These rectangles are grouped into 16 leaf level nodes, numbered 1 to 16. The bounding boxes of each set of rectangles stored at a node serve as rectangles to be stored at the next level. Thus, leaf level nodes 1 through 4 are placed in node 17 in level 1. The root node contains the 4 level 1 nodes: 17, 18, 19, and 20.

To perform a query Q, all rectangles (internal or not) that intersect the query region must be retrieved. This is accomplished with a simple recursive procedure that starts at the root and (possibly) follows several paths along the tree. A node is processed by first computing all rectangles stored at that node that intersect Q. If the node is an internal node the subtrees pointed to by the retrieved rectangles (if any) are processed recursively. Otherwise, the node is a leaf node and the retrieved rectangles are simply reported. Consider, for example, query Q in Figure 1. After processing the root node, we determine that nodes 19 and 20 of

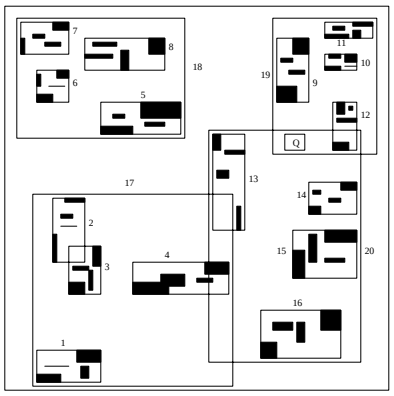

Figure 1: A sample R-tree. Input rectangles are shown solid.

level 1 must be searched. The search proceeds in both of these nodes. In both cases we find the query region does not intersect any rectangles within the two nodes and both subqueries are terminated.

From the previous discussion it is clear that, in general, rectangles at any level may overlap and that the R-tree for a set of rectangles is by no means unique. Ideally, one would want to construct (and maintain) an R-tree so as to minimize the average query time. Kamel and Faloutsos [6] derive an analytical formula for the number of rectangles retrieved for uniformly distributed queries of a fixed size, and show that this number is proportional to the sum of areas and sum of perimeters of stored rectangles.

Guttman provides efficient but far from optimal routines for insertion and deletion of rectangles. Unfortunately, building an R-tree by inserting one object at a time as specified in [4] has several disadvantages: (a) high load time, (b) sub-optimal space utilization, and (c) poor R-tree structure possibly necessitating many nodes to be searched to satisfy a query. For applications where the data is available a priori (as is the case in many scientific and geographic applications, index creation required by a DBMS for query processing, or when loading tape resident data) more efficient offline algorithms can be invoked to optimize the structure of the tree in such a way that fewer nodes need to be accessed to satisfy a query. Such *bulk loading algorithms* were first proposed by Roussopoulos [14] and later by Kamel and Faloutsos [6]. These algorithms result in trees with near 100% space utilization and improved query efficiency. Although these algorithms result in superior R-tree structures relative to inserting one tuple at a time, both have serious problems. The algorithm of [14] results in poor query performance for query regions with a large extent in the sorted dimension, that of [6] is unnecessarily complicated.

In this work we have developed three new bulk-loading algorithms and two new dynamic algorithms. Descriptions of the algorithms and results are presented in section 4.

4 Algorithms and Results

In this section we describe the new algorithms and highlight key results. A more complete description of the results can be found in the papers cited. We have developed three techniques for static data: STR, TGS, and Post-Opt. We have developed two new techniques for dynamic data: Shift, and MIX. We describe each below and include a small set of results for each.

4.1 Static Data Packing Background

We start first with an overall framework for loading (or packing) R-trees and then explicitly describe each policy. We first describe previous packing algorithms including Nearest-X [14], Hilbert [6] and then our new Sort-Tile-Recursive (STR) algorithm. Readers interested in more detailed descriptions should refer to [14, 6, **?**].

All of the algorithms use a similar framework. In the following text we assume that the data file consists of r rectangles and that each R-Tree node can hold n rectangles.

The general process is similar to building a B-tree from a collection of keys by creating the leaf level first and then creating each successively higher level until the root node is created [13].

General Algorithm:

1. Preprocess the data file so that the r rectangles are ordered in $\lceil r/n \rceil$ consecutive groups of n rectangles, where each group of n is intended to be placed in the same leaf level node. Note that the last group may contain fewer than n rectangles.
2. Load the $\lceil r/n \rceil$ groups of rectangles into pages and output the (MBR, page-number) for each leaf level 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 three algorithms differ only in how the rectangles are ordered at each level.

Nearest-X (NX):

This algorithm was proposed in [14]. The rectangles are sorted by x-coordinate. No details are given in the paper so we assume that the x-coordinate of the rectangle's center is used. The rectangles are then packed into the nodes, in groups of size n, using this ordering.

Hilbert Sort (HS):

A fractal based algorithm was proposed in [6]. The algorithm orders the rectangles using the Hilbert (fractal) space filling curve. The center points of the rectangles are sorted based on their distance from the origin, measured along the Hilbert Curve. This determines the order in which the rectangles are placed into the nodes of the R-Tree.

Kamel and Faloutsos only provide details on how to handle integer coordinates, which can be extended to arbitrary floating point values as described below.

Floating point numbers are usually stored as a sign, signed exponent, and a mantissa (which may or may not be normalized). Since the exponent determines the starting position of the mantissa relative to the binary point, all floating point numbers could be represented using $2^{\text{sizeof}(\textit{Exponent})} + \text{sizeof}(\textit{Mantissa})$ bits. For example, for 32-bit float numbers native in the Sun Sparc architecture,

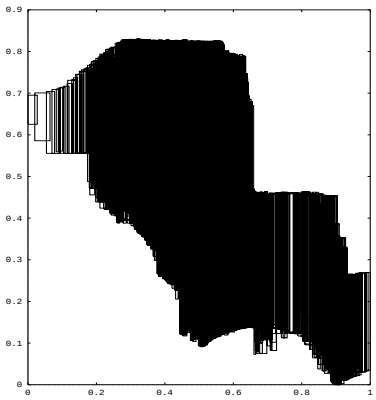

Figure 2: Leaf Bounding Rectangles for Long Beach Data using NX

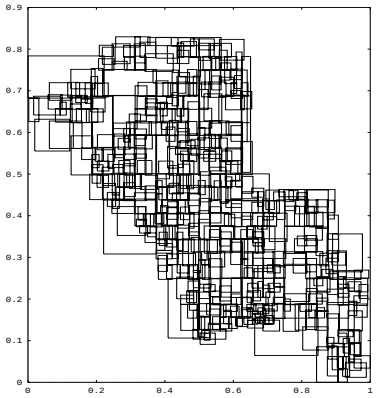

Figure 3: Leaf Bounding Rectangles for Long Beach Data using HS

$2^8 + 23$ bits would required. (This is a conceptual representation only. Coordinates are not stored in this form.) Once the numbers are viewed using this representation, it is clear that the method used for integers can be applied.

We now briefly describe the processing of a 2-dimensional data set. Consider a grid of size $2^{\text{sizeof}(Exponent)+\text{sizeof}(M}$ The Hilbert Curve for this grid is used to produce the packing order of the rectangles. The first bit of the x- and y-coordinates of a point determine which quadrant contains it. Successive bits determine which successively smaller subquadrants contain the point. When two center points (x_1, y_1) and (x_2, y_2) need to be compared, the bits of each coordinate are examined until it can be determined that one of the points lies in a different subquadrant than the other (one can use the sense and rotation tables described in [6] to accomplish this task). The information gathered is used to decide which point is closer to the origin (along the Hilbert Curve). Conceptually, the process computes bit positions, one at a time, until discrimination is possible. In practice, one does not store or compute all bit values on the hypothetical grid.

4.2 Sort-Tile-Recursive (STR):

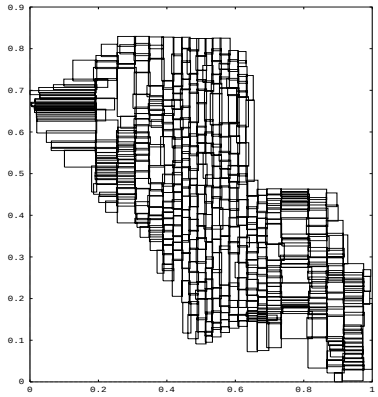

Figure 4: Leaf Bounding Rectangles for Long Beach Data using STR

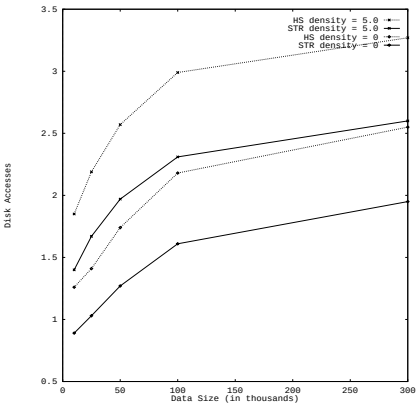

Figure 5: Disk Accesses vs. Data Size for Point Queries on Synthetic Data, Buffer Size 10

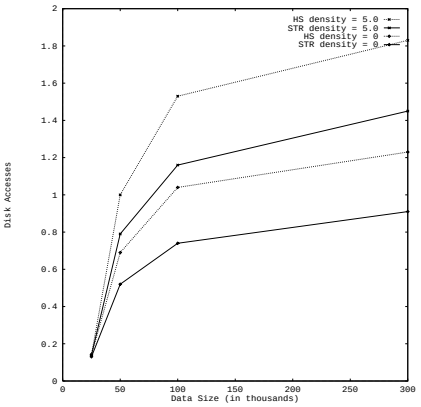

Figure 6: Disk Accesses vs. Data Size for Point Queries on Synthetic Data, Buffer Size 250

Consider a k-dimensional data set of r hyper-rectangles. A hyper-rectangle is defined by k intervals of the form $[A_i, B_i]$ and is the locus of points whose i-th coordinate falls inside the i-th interval, for all $1 \leq i \leq k$.

STR is best described recursively with $k = 2$ providing the base case. (The case $k = 1$ is already handled well by regular B-trees.) Accordingly, we first consider a set of rectangles in the plane. The basic idea is to "tile" the data space using $\sqrt{r/n}$ vertical slices so that each slice contains enough rectangles to pack roughly $\sqrt{r/n}$ nodes. Once again we assume coordinates are for the center points of the rectangles. Determine the number of leaf level pages $P = \lceil r/n \rceil$ and let $S = \lceil \sqrt{P} \rceil$. Sort the rectangles by x-coordinate and partition them into S vertical slices. A slice consists of a run of $S \cdot n$ consecutive rectangles from the sorted list. Note that the last slice may contain fewer than $S \cdot n$ rectangles. Now sort the rectangles of each slice by y-coordinate and pack them into nodes by grouping them into runs of length n (the first n rectangles into the first node, the next n into the second node, and so on).

The case $k > 2$ is is a simple generalization of the approach described above. 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 $n \cdot \lceil P^{\frac{k-1}{k}} \rceil$ consecutive hyper-rectangles from the sorted list. Each slab is now processed recursively using the remaining $k - 1$ coordinates (i.e., treated as a $k - 1$-dimensional data set).

To aid in visualizing the result of these packing algorithms, consider the leaf level nodes obtained by using the Long Beach Tiger data set assuming 100 rectangles fit per node. Figures 2, 3, and 4 show the resultant leaf level MBR for the same data set for each of the three algorithms. Note the vertical slices in Figure 4 for the STR packing algorithm.

We now consider results for synthetic data. Additional results can be found in [8]. We consider buffer sizes of 10 and 250 pages. In Table 1 we show what percent of the R-tree fits in the buffer for the different sizes of data sets. The first column is the number of rectangles, the second is the number of R-tree pages (including non-leaf) assuming 100 rectangles per page, the third is the percent of the R-tree a buffer of 10 pages can hold, and the fourth is the percent a buffer of 250 pages can hold. We do not consider a data size of 10,000 for a buffer of 250 pages as the entire R-tree fits in the buffer.

Figures 5 and 6 plot the number of disk accesses versus data set size (in thousands of rectangles) for point queries using a buffer size of 10 and 250 pages respectively. The top two curves in each figure are for a

Data Size	R-Tree Pages	Buffer = 10	Buffer = 250
10,000	101	9.90%	100%
25,000	254	3.94%	98.43%
50,000	506	1.98%	49.41%
100,000	1011	0.99%	24.73%
300,000	3031	0.33%	8.25%

Table 1: Percent of R-Tree Held By Buffer

	Point Data					Region Data, Density = 5.0				
Data Size	STR	HS	NX	HS/STR	NX/STR	STR	HS	NX	HS/STR	NX/STR
Point Queries										
10	0.89	1.26	0.87	1.42	0.99	1.40	1.85	3.52	1.32	2.51
25	1.03	1.41	1.04	1.38	1.01	1.67	2.19	6.11	1.31	3.67
50	1.27	1.74	1.27	1.37	1.00	1.97	2.57	8.43	1.30	4.28
100	1.61	2.18	1.57	1.35	0.97	2.31	2.99	12.45	1.29	5.39
300	1.95	2.55	1.83	1.31	0.94	2.60	3.27	19.70	1.26	7.56
Region Queries, Query Region = 1% of Data										
10	3.27	3.99	10.86	1.22	3.33	4.25	4.97	13.78	1.17	3.24
25	6.85	8.00	26.61	1.17	3.89	8.53	9.87	31.44	1.16	3.69
50	11.48	12.81	50.64	1.12	4.41	13.12	14.55	57.52	1.11	4.38
100	18.21	19.93	98.47	1.09	5.41	20.40	22.14	108.37	1.09	5.31
300	41.46	44.02	290.05	1.06	7.00	44.73	47.26	307.38	1.06	6.87
Region Queries, Query Region = 9% of Data										
10	11.73	13.02	26.86	1.11	2.29	13.57	14.80	29.91	1.09	2.20
25	26.40	28.07	67.26	1.06	2.55	29.01	30.76	72.17	1.06	2.49
50	46.20	48.74	131.96	1.05	2.86	49.48	51.97	48.74	1.05	0.98
100	84.54	87.51	261.35	1.04	3.09	89.25	92.18	271.58	1.03	3.04
300	229.75	234.82	779.96	1.02	3.39	237.42	242.41	797.94	1.02	3.36

Table 2: Number of Disk Accesses, Synthetic Data, Buffersize = 10

data density of 5 (i.e., the expected sum of the areas of the input rectangles equals 5) and the bottom two curves are for a density of 0 (i.e., point data). The legends in the figures show the ordering of the lines from top to bottom. The solid lines are for STR and the dashed lines are for HS. For a buffer of size 10, HS requires 31 - 42% more disk accesses than STR for point data, and 26 - 32% more disk access for region data of density 5. For a buffer of size 250, HS requires 33 - 41% more disk access than STR for point data, and 26 - 32% more disk access for region data of density 5. Note, that for the 25,000 rectangle data set almost the entire tree fits and hence the comparison is not particularly meaningful.

More exhaustive results are presented in Tables 2 and 3 for buffer sizes of 10 and 250 pages, respectively. The first column is the number of data items in thousands, the second through fourth columns are the number of disk accesses to satisfy the query for STR, HS, and NX on point data, the fifth and sixth columns are the ratio of HS and NX relative to STR for point data, and columns 7-11 are the same as 2-6 but for region data of density 5. Note that NX is not competitive except for point queries on point data, and, as expected, the difference between STR and HS diminishes as the query size increases.

	Point Data					Region Data, Density = 5.0				
Data Size	STR	HS	NX	HS/STR	NX/STR	STR	HS	NX	HS/STR	NX/STR
Point Queries										
25	0.13	0.14	0.13	1.03	1.00	0.14	0.14	0.20	1.05	1.43
50	0.52	0.69	0.52	1.33	1.01	0.79	1.00	3.85	1.27	4.88
100	0.74	1.04	0.77	1.41	1.04	1.16	1.53	8.05	1.32	6.95
300	0.91	1.23	0.92	1.35	1.01	1.45	1.83	16.98	1.26	11.67
Region Queries, Query Region = 1% of Data										
25	0.16	0.17	0.49	1.06	3.05	0.17	0.19	1.01	1.14	6.04
50	4.74	5.30	26.24	1.12	5.54	5.57	6.15	29.85	1.10	5.36
100	12.14	13.27	76.60	1.09	6.31	13.84	14.94	84.53	1.08	6.11
300	36.72	38.92	279.67	1.06	7.62	39.84	42.04	296.80	1.06	7.45
Region Queries, Query Region = 9% of Data										
25	0.20	0.21	1.29	1.08	6.51	0.25	0.25	2.88	1.00	11.57
50	20.11	21.20	76.11	1.05	3.78	22.12	23.10	81.78	1.04	3.70
100	61.78	64.18	228.98	1.04	3.71	65.52	68.12	239.43	1.04	3.65
300	218.61	224.09	769.74	1.03	3.52	226.77	231.62	787.30	1.02	3.47

Table 3: Number of Disk Accesses, Synthetic Data, Buffersize = 250

4.3 Top-down Greedy Splitting (TGS)

Our second algorithm, TGS, differs from STR in that it explores more candidate partitioning choices and builds the tree top-down using a data adaptive step which rearranges the database objects under a node in construction. Trees built by TGS achieve up to three times improvement in query performance.

The algorithm was motivated by two key ideas:

1. Minimize first the top levels since the potential for cost reduction is higher.
2. Consider all partitions induced by guillotine cuts such that resulting sub-trees are fully packed.

TGS recursively applies a *basic split step* which partitions a set of n rectangles into two subsets by a cut orthogonal to an axis. Such a cut must meet two conditions: (a) the cost of some objective function $f(r_1, r_2)$, where r_1 and r_2 are the MBRs of the two ensuing parts, is minimum, and (b) one subset has a cardinality $i \cdot S$ for some i where S is fixed per level so that the resulting subtrees are packed. The recursion is applied to both subsets until there is one subset per child.

A description of the basic split process is shown in Figure 7. Based on the extreme values defining the rectangle, we considered orderings on: min., max., both, center. A more detailed description of the algorithm, load time analysis, and results can be found in [2].

Our comparison metric is the ratio of the number of disk accesses needed for HS or STR over the number of disk accesses needed for TGS. The maximum number of entries per node M is 100. We present results for buffer sizes of 10, 25, 50, 100, and 250 pages, one table column per each value.

TGS can consider any cost function and several orderings. We present the results for area as cost function since there was no significant difference when using the weighted perimeter as an objective function [6].

```
Let n = number of input rectangles
Let S = maximum number of rectangles per subtree
Let M = maximum number of entries per node
Let f(r1, r2) be the "user-supplied" cost function
If n ≤ S return {stop condition}
For each dimension d
    For each ordering considered in this dimension d
        For i from 1 to ⌈n/M⌉ − 1
            Let B0 = MBR of first i · S rectangles
            Let B1 = MBR of the other rectangles
            Remember i if f(B0, B1) is better valued
Split input set and orderings at best position.
```

Figure 7: The basic split step

	Performance ratio HS/TGS				
data	10	25	50	100	250
VLSI	2.09	1.98	1.90	1.84	1.74
TILE	2.79	3.03	2.85	2.72	2.32
UNIF	1.32	1.32	1.33	1.33	1.34

Table 4: Point query, Hilbert vs TGS

Only center point value ordering is presented since results do not significantly differ from using minimum and maximum ordering.

Two general trends were observed. First, TGS outperforms STR or HS under significant skew in location, area, or aspect ratio. As seen in Tables 4 and 5, TGS achieves an improvement of up to 3 times when dealing with data with skew. However there is little gain when dealing with uniform data. Modifications on data set distributions for locations, areas, aspect ratios lead to similar conclusions.

4.4 Post Optimization

In this section we describe our new post-optimization algorithm. The algorithm uses a node restructuring process to improve R-tree structure. We first describe the node restructuring process, then the full post-optimization algorithm.

Node Restructuring

The goal of node restructuring is to incrementally improve global R-tree structure. To this end, node restructuring is based on the following principles:

- Maintain or increase node utilization by merging nodes when possible. This is especially useful when starting from trees built by dynamic insertion algorithms.

	Performance ratio STR/TGS				
data	10	25	50	100	250
VLSI	2.16	2.14	2.11	2.10	1.98
TILE	2.29	2.44	2.32	2.23	1.93
UNIF	0.99	0.98	0.99	0.99	1.02

Table 5: Point query, STR vs TGS

- Redistribute node entries among siblings in such a way so as to reduce expected query cost. To do this we incrementally reduce the sum of areas/perimeters of a given node and its siblings by redistributing the data entries according to the heuristic process described below.
- When deciding which siblings to exchange entries with, chose nodes whose MBR overlaps the MBR of the node being restructured. This tends to limit the amount of work done.
- Before redistributing among siblings, first make sure siblings are as "close" as possible by restructuring the parent node. Consider Figure 8. If node r_x is chosen for restructuring and the parent is not first restructured, only r_0, r_1, r_2, and r_3 would be considered for entry redistribution. Conversely, if the parent is restructured first, r_x may end up in the same node as b_3 thus allowing a more beneficial restructuring. Furthermore, restructuring the parent allows local changes to be propagated more globally.

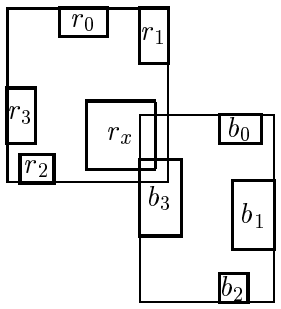

Figure 8: Restructuring Parent First

These principles are organized together to form the node restructuring algorithm shown in Figure 9.

We have investigated the effect of several split algorithms for use in the basic node restructuring process. We considered the following:

- Pseudo-Optimal: This is the split algorithm proposed in [2]. It attempts to optimally find the split that minimizes the given objective function. It may omit some candidates when cardinality constraints are included, as done in this paper, and hence may find a sub-optimal solution. For the

```
restructure( node n )
{
    if n is not the root then
        restructure( parent(n) )
    mark all sibling(n) and n as clean
    foreach node s that is a clean overlapping sibling(n)
        if |s| + |n| ≤ M then
            Let n = s ∪ n          { merge n, s }
            remove entry for s in parent(n)
            go out of the for loop
        else
            (n1, n2) = split( n ∪ s )
            Let n = n1
            make entry for parent(s) point to n2
            mark n2 as dirty
}
```

Figure 9: The basic node restructuring process

objective function we use the area and weighted perimeter formula proposed by Kamel et. al [6] and Pagel et. al [11].

- R-star: Node splitting used for the r*-tree [1].
- Axis-Sweep: Split based on a sweep line along the coordinates of rectangle centers over all axes. The rectangles are sorted on each axis and a partitioning at each point in the sorted list is considered. A partitioning occurring at a given rectangle's center coordinate divides the rectangles into one subset whose center points come before the given rectangles center coordinate and the remaining rectangles go into the other subset. The partitioning over all axes that minimizes the objective function is chosen.
- Hilb-Sweep: Same as before, but Hilbert values are used to guide the sweep-line instead of coordinates.

We have found that the pseudo-optimal split [2] results in significantly better performance than the other split function choices.

Our post-optimization algorithm consists of a sequence of node restructurings. In [3] we show that for post-optimizing a tree of n nodes, a sequence of n restructurings suffices to get a good query performance improvement. We considered six different ways of selecting the next node to restructure. In general we found random selection to be competitive with more sophisticated weight driven and query driven approaches.

We now present results assuming the data set is static and hence we need not be concerned with insert/deletes after the initial R-tree is created. Our method is to first build the tree for the given data set using a given loading method and then post-optimize the tree by applying a series of restructurings.

In Table 6 we present the disk access ratios for post-optimization of the dynamic-Hilbert, packed-Hilbert, and STR loading algorithms. We include the dynamic-Hilbert algorithm to show that post-optimization can be applied to any tree built by any loading algorithm, not just static data packing algorithms.

As can be seen from the table, post-optimization can decrease the number of query disk accesses by a factor of 2. When averaged over all data sets considered post-optimization reduced the number of disk accesses by 40% and 27% respectively for packed-Hilbert and STR trees. As shown in [3], the cost for the post-optimization is about the same as the initial loading using STR and twice that of Hilbert. Thus, by quadrupling (doubling) the load cost, the average query cost can be reduced by 40% (27%) for the Hilbert (STR) algorithm. Since the data set is only loaded once the initial loading cost is amortized over subsequent queries resulting in better overall performance.

Note that post-optimization does not improve STR for the uniform data sets. This is because for the uniform data set STR creates a packing with no leaf level intersections and hence the node restructurings (which repartition the restructured node with all overlapping siblings) do nothing. We did try a different optimization algorithm where each node restructure reparations with the nearest sibling only. This optimization did improve STR for uniform data but did not result in as significant of improvements for the other data sets as repartitioning with all of the intersecting siblings.

In Table 7 we present the results for the same experiments assuming region queries of size (.02 x .02). Note, the trends are the same but the magnitude of improvement is reduced.

data	dynamic Hilbert	packed Hilbert	STR
cfd	1.36	1.21	1.13
cluster	1.53	1.36	1.11
tiger	1.53	1.38	1.08
unif.0	1.29	1.24	1.00
unif.1	1.31	1.23	1.00
vlsi.J	2.08	1.78	2.13
vlsi.N	1.81	1.57	1.47
avg.	1.56	1.40	1.27

Table 6: Point query static improvement

data	dynamic Hilbert	packed Hilbert	STR
cfd	1.08	1.06	1.04
cluster	1.36	1.23	1.11
tiger	1.33	1.22	1.05
unif.0	1.19	1.16	1.00
unif.1	1.22	1.15	1.00
vlsi.J	1.71	1.50	1.72
vlsi.N	1.55	1.38	1.31
avg.	1.34	1.24	1.18

Table 7: .02 x .02 query static improvement

4.5 Dynamic MIX

The MIX algorithm is used for building dynamic R-trees. It uses the same node re-structuring algorithm described in section 4.4. After an insertion, a restructuring is invoked with probability p. The node to be restructured is not the node just inserted into, but rather is chosen as described above for the Post-Optimization process. As shown in [3], we have found that a value of $p = 0.04$ achieves a good balance between overhead and good tree structure. These restructurings only increase the insertion cost by at most 10%. Furthermore, some of these restructurings may occur during idle periods and hence reduce the overhead of the restructurings to less than 10%. Any dynamic node insertion algorithm (such as Guttman's quadratic, R*, Hilbert) can be used. For our tests, we use Guttman's insertion with quadratic split.

We now present results for dynamic data and compare with existing dynamic R-tree algorithms as well as compared to existing static data algorithms. We assume the minimum node utilization during a split is 40%, and the objective function value for perimeter weighting is 0.2. Furthermore, we assume the probability that an insertion triggers a node restructuring is 0.04. Note, the total number of nodes restructured during building of the the tree is roughly twice the number of nodes in the final tree. Hence, the restructuring overhead is about double that of the post-optimization processes. Regardless, the restructurings require less than 10% additional I/O cost relative to the normal dynamic insertion algorithms.

data	dynamic Hilbert	packed Hilbert	STR	GUT	post-opt GUT
cluster	1.37	1.32	1.08	3.19	0.89
tiger	1.40	1.38	1.04	1.45	0.89
unif.0	1.18	1.18	0.89	11.43	1.00
unif.1	1.10	1.09	0.82	3.56	0.92
vlsi.J	1.71	1.58	1.92	1.93	0.86
vlsi.N	1.35	1.30	1.21	2.66	0.87
avg.	1.35	1.31	1.16	4.04	0.91

Table 8: MIX point query disk access ratios

In Table 8 we present THE QUERY disk access ratios, where the denominator is the average number of disk access for MIX. Comparing first with other dynamic algorithms we find that on average MIX requires 35% (304%) fewer disk accesses than Hilbert (Guttman). For some data sets the improvement is significantly higher. Thus, MIX significantly outperforms the current state of the art.

On average, MIX is even superior to the static packing algorithms such as static-Hilbert and STR even though MIX is not as good as STR only for uniform point data. Finally, we compare MIX to post-optimization of a tree built using Guttman's quadratic split. We see that despite MIX doing almost double the number of node restructurings, the post-optimization algorithm performs best. Note, if we apply post-optimization to the MIX tree it too converges to the performance of the post-optimized Guttman tree.

In brief, MIX can be used to improve existing dynamic algorithms so that their performance is superior to existing packing algorithms. We have assumed that on average 4% of the insertions trigger a node restructure.

4.6 Dynamic SHIFT

Our dynamic SHIFT algorithm utilizes a new splitting algorithm that is probably optimal when no cardinality constraints are considered. In the next section we describe the split algorithm. The reading is rather technical and formal and not necessary for understanding the SHIFT algorithm. The reader not interested in these details should skip to section 4.6.2.

4.6.1 Optimal Splitting: The Bipartition Algorithm

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.

Table 9: 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

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.

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 9, 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. $\square$

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. $\square$

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

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 10.

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 11. 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.

```
{Preprocessing}
For each one u of {x, y, X, Y}
    Sort all the C_u s
    Incrementally build the MBR complementary wrt C_u as the MBR
    of previous C_u in the ordering and said C'_u s 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 10: The basic optimal bipartition algorithm

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 10 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. 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.

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 10. However, a few changes to

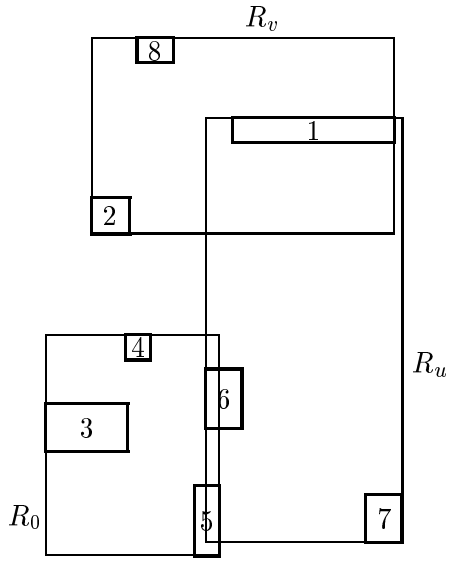

Figure 11: R_u is the MBR complementary to R_0 wrt C_{X,R_0}, and R_v is so wrt C_{Y,R_0}. The values C_{x,R_0} and C_{y,R_0} are not considered since R_0 shares those values with R_s.

the basic algorithm are sufficient to compute bipartitions with cardinality constraints while preserving the quadratic running 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 12 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 13):

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

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

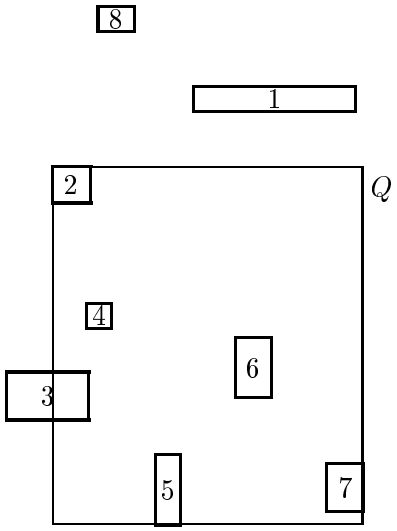

Figure 12: For corner $C_{x,2}, C_{Y,2}$ of rectangle 2 (upper left corner), Q is the MBR containing rectangles 2, 5 and 7. The corner cardinality is 5. Note that the x, Y corner of rectangle 3 is not counted as it is outside Q.

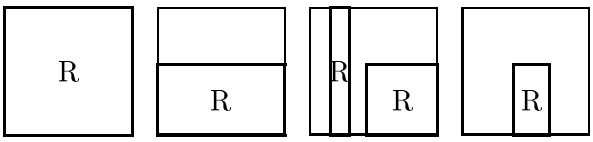

Figure 13: MBR R with 4,3,2, and 1 R_s defining values.

```
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 14: Computing corners cardinality

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 $Card[C_u, C_v]$ 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 10) 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 14 (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 $Card[C_x, C_y]$, 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 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.

4.6.2 The SHIFT Framework

We now describe our insertion method for building 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 [7] 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 [14, 6, 8] 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 [6].) 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.

4.6.3 SHIFT 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 [4] under two different splitting policies: Guttman's quadratic splitting (GUT) and optimal splitting (OPT).

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 [1]. 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 10 we present results for the tiger Long Beach data set (a data set of about 50,000 line segments representing roads in Long Beach California). 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

Table 10: 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

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.

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 [1], using packing algorithms [14, 6, 8], or taking into consideration more global properties of the tree such as the forced reinserts of the R* tree [1] and the Hilbert ordering structure of the Hilbert tree [7], or the restructuring of our new SHIFT algorithm.

5 Technology Transfer

The following code has been delivered to MetaComp.

- Basic R-tree structure.
- STR bulk loading routines.
- Dynamic insertion routines using a simplified variant of the $R^\star$ splitting algorithm. Note, we show in our paper that the query performance improvement obtained from optimal splitting does not warrant the split cost, hence we delivered the simpler solution.
- A generalization of the code to accommodate any number of dimensions.

6 Networking

Attendance at the following conferences was made possible in part by this CASI grant:

- IEEE International Conference on Data Engineering (ICDE97), April, 1997, Birmingham, England.
- IEEE International Conference on Data Engineering (ICDE98), February 1998, Orlando.

- ACM Symposium on Geographic Information Systems, November 1998, Washington DC.
- International Conference on Very Large Databases, August 1998, New York City

7 Publications

The following publications were made possible in part by support from this grant:

(1) Leutenegger, S.T., Lopez, M.A., "The Effect of Buffering on the Performance of R-Trees", to appear in IEEE Transactions on Knowledge and Data Engineering (TKDE) early 1999. This paper is an expanded version of publication (4) below.

(2) Garcia, Y.J, Lopez, M.A., Leutenegger, S.T., "A Greedy Algorithm for Bulk Loading R-Trees", in Proc. of 6th ACM Symposium on Geographic Information Systems (GIS'98), 2 page poster paper, Nov 1998.

(3) Garcia, Y.J, Lopez, M.A., Leutenegger, S.T., "On Optimal Node Splitting for R-trees", in Proc. of VLDB 98, New York NY, Aug 1998

(4) Leutenegger, S.T., Lopez, M.A., "The Effect of Buffering on the Performance of R-Trees", in the Proc. of the 14th International Conference on Data Engineering (ICDE 98), Orlando Florida, Feb 1998, pp. 164-171

(5) Leutenegger, S.T., Lopez, M.A., Edgington, J.M., "STR: A Simple and Efficient Algorithm for R-Tree Packing", in the Proc. of the 13th International Conference on Data Engineering (ICDE 97), Birmingham England, Apr 1997, pp. 497-506

The following papers will be submitted for publication:

(6) Garcia, Y.J., Lopez, M.A., Leutenegger, S.T., "Master-Client R-trees: A New Parallel R-tree Architecture",

(7) Schnitzer, B., Leutenegger, S.T., "Post-optimization and Incremental Refinement of R-trees",

8 Funding

This CASI grant provided the "seed" for the following NSF grant:

1997-2000 *Geometric Techniques for Multidimensional Databases* **$333,743**
S.T. Leutenegger and M.A. Lopez)
National Science Foundation

In addition, this CASI grant contributed to the successful funding of the following NSF grant:

1998-2002 *Integration of Computational and Data Access Scheduling in NOWs* **$224,863**
S.T. Leutenegger
National Science Foundation Career Award

References

[1] N. Beckmann, H.P. Kriegel, R. Schneider, and B. Seeger. The $r^\star$-tree: an efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD*, pages 323–331, may 1990.

[2] Y. Garcia, M.A. Lopez, and S.T. Leutenegger. On optimal node splitting for r-trees. In *Proc. of VLDB 98*, New York, NY, August 1998.

[3] Y. Garcia, M.A. Lopez, and S.T. Leutenegger. Post-optimization and incremental refinement of r-trees. submitted for publication, 1998.

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

[5] K.C. Sevcik J. Nievergelt, H. Hinterberger. The grid file: An adaptable, symetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, March 1984.

[6] I. Kamel and C. Faloutsos. On packing r-trees. In *Proc. of Second Int. Conf. on Information and Knowledge Management (CIKM)*, Washington, D.C., November 1993.

[7] I. Kamel and C. Faloutsos. Hilbert r-tree: an improved r-tree using fractals. In *Proc. of VLDB Conference,*, pages 500–509, Santiago, Chile, September 1994.

[8] S.T. Leutenegger, M.A. Lopez, and J.M. Edgington. Str: A simple and efficient algorithm for r-tree packing. In *Proc. of the 14th International Conference on Data Engineering (ICDE 97)*, pages 497–506, 1997.

[9] D. Lomet and B. Salzberg. The hb-tree: A multiatrribute indexing method with good guaranteed performance. *ACM Trans. on Database Systems*, 15(4):625–658, 1990.

[10] J. Orenstein. Spatial query processing in an object-oriented database system. In *Proc. ACM SIGMOD*, pages 326–335, may 1986.

[11] B-U. Pagel, H-W. Six, H. Toben, and P. Widmayer. Towards an analysis of range query performance in spatial data structures. In *Proc. ACM PODS*, pages 214–221, May 1993.

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

[13] A.L. Rosnberg and L. Snyder. Time and space optimality in b-trees. *ACM Transactions on Database Systems*, 6(1), 1981.

[14] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed r-trees. In *Proc. ACM SIGMOD*, pages 17–31, Austin, Texas, May 1985.