R-TREES: A DYNAMIC INDEX STRUCTURE FOR SPATIAL SEARCHING

by

Antonin Guttman and Michael Stonebraker

Memorandum No. UCB/ERL M83/64

14 October 1983

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Research sponsored by the National Science Foundation Grant ECS-8300463 and the Air Forces Office of Scientific Research Grant AFOSR-83-0254.

1. Introduction

Spatial data objects often cover areas in multi-dimensional spaces and are not well represented by point locations. For example, map objects like counties, census tracts etc. occupy regions of non-zero size in two dimensions. A common operation on spatial data is to search for all objects in an area. An example would be to find all the counties that have land within 20 miles of a particular point. This kind of spatial search occurs frequently in computer aided design (CAD) and geo-data applications. In such applications it is important to be able to retrieve objects efficiently according to their spatial location.

An index based on objects' spatial locations is desirable, but classical one-dimensional database indexing structures are not appropriate to multi-dimensional spatial searching. Structures based on exact matching of values, such as hash tables, are not useful because a range search is required. Structures using one-dimensional ordering of key values, such as B-trees and ISAM indexes, do not work because the search space is multi-dimensional.

A number of structures have been proposed for handling multi-dimensional point data, and a survey of methods can be found in [5]. Cell methods [4,8,16] are not good for dynamic structures because the cell boundaries must be decided in advance. Quad trees [7] and k-d trees [3] do not take paging of secondary memory into account. K-D-B trees [13] are designed for paged memory but are only useful for point data. The use of index intervals has been suggested in [15], but this method cannot be used in multiple dimensions. Corner stitching [12] is an example of a structure for two-dimensional spatial searching suitable for data objects of non-zero size, but it assumes homogeneous primary memory and is not efficient for random searches in very large collections of data. Grid files [10] handle non-point data by mapping each object to a point in a higher-dimensional space. In this paper we describe an alternative structure called an R-tree which represents data objects by intervals in several dimensions.

Section 2 outlines the structure of an R-tree and Section 3 gives algorithms for searching, inserting, deleting, and updating. Results of R-tree index performance tests are presented in Section 4. Section 5 contains a summary of our conclusions.

2. R-Tree Index Structure

An R-tree is an index structure for n-dimensional spatial objects analogous to a B-tree [2, 6]. It is a height-balanced tree with records in the leaf nodes each containing an n-dimensional rectangle and a pointer to a data object having the rectangle as a bounding box. Higher level nodes contain similar entries with links to lower nodes. Nodes correspond to disk pages if the structure is disk-resident, and the tree is designed so that a small number of nodes will be visited during a spatial search. The index is completely dynamic; inserts and deletes can be intermixed with searches and no periodic reorganization is required.

A spatial database consists of a collection of records representing spatial objects, and each record has a unique identifier which can be used to retrieve it. We approximate each spatial object by a bounding rectangle, i.e. a collection of intervals, one along each dimension:

$$I = (I_0, I_1, \ldots, I_{n-1})$$

where n is the number of dimensions and I_i is a closed bounded interval $[a,b]$ describing the extent of the object along dimension i. Alternatively I_i may have one or both endpoints equal to infinity, indicating that the object extends outward indefinitely.

Leaf nodes in the tree contain index record entries of the form

$$(I,\ tuple\text{-}identifier)$$

where *tuple-identifier* refers to a tuple in the database and I is an n-dimensional rectangle containing the spatial object it represents. Non-leaf nodes contain entries of the form

$$(I,\ child\text{-}pointer)$$

where *child-pointer* is the address of another node in the tree and I covers all rectangles in the lower node's entries. In other words, I spatially contains all data objects indexed in the subtree rooted at I's entry.

Let M be the maximum number of entries that will fit in one node and let $m \leq \frac{M}{2}$ be a parameter specifying the minimum number of entries in a node. An R-tree satisfies the following properties:

(1) Every leaf node contains between m and M index records unless it is the root.

(2) For each index record $(I, tuple\text{-}identifier)$ in a leaf node, I is the smallest rectangle that spatially contains the n-dimensional data object represented by the indicated tuple.

(3) Every non-leaf node has between m and M children unless it is the root.

(4) For each entry $(I, child\text{-}pointer)$ in a non-leaf node, I is the smallest rectangle that spatially contains the rectangles in the child node.

(5) The root node has at least two children unless it is a leaf.

(6) All leaves appear on the same level.

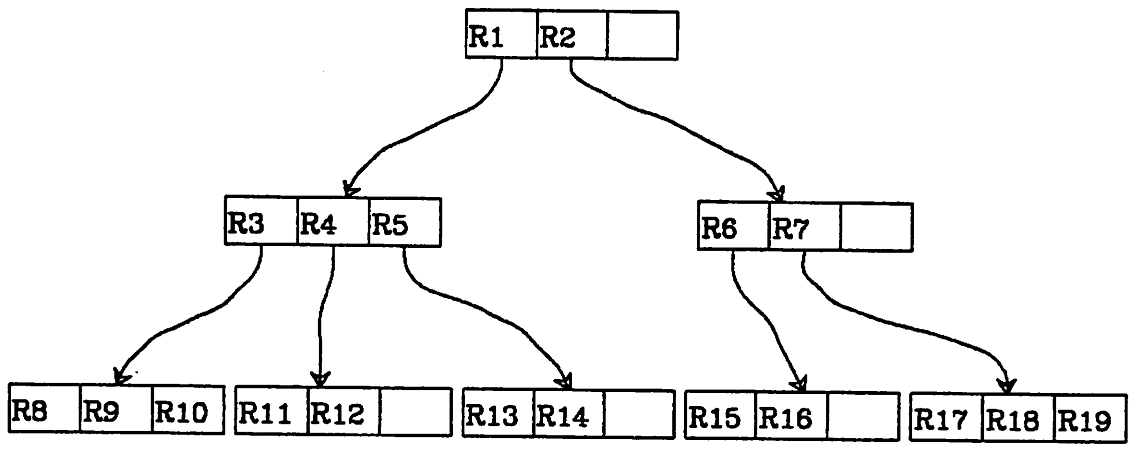
Figure 2.1a and 2.1b show an example R-tree structure and the geometric forms it represents.

The height of an R-tree containing N index records is at most $\lceil \log_m N \rceil$, because the branching factor of each node is at least m. The maximum number of nodes is $\left\lceil \frac{N}{m} \right\rceil + \left\lceil \frac{N}{m^2} \right\rceil + \cdots + 1$. Worst-case space utilization for all nodes except the root is $\frac{m}{M}$. Nodes will tend to have more than m entries, and this will decrease tree height and improve space utilization. If nodes have more than 3 or 4 entries the tree will be very wide, and almost all the space will be used for leaf nodes containing index records. The parameter m can be varied as part of performance tuning, and different values are tested experimentally in Section 4.

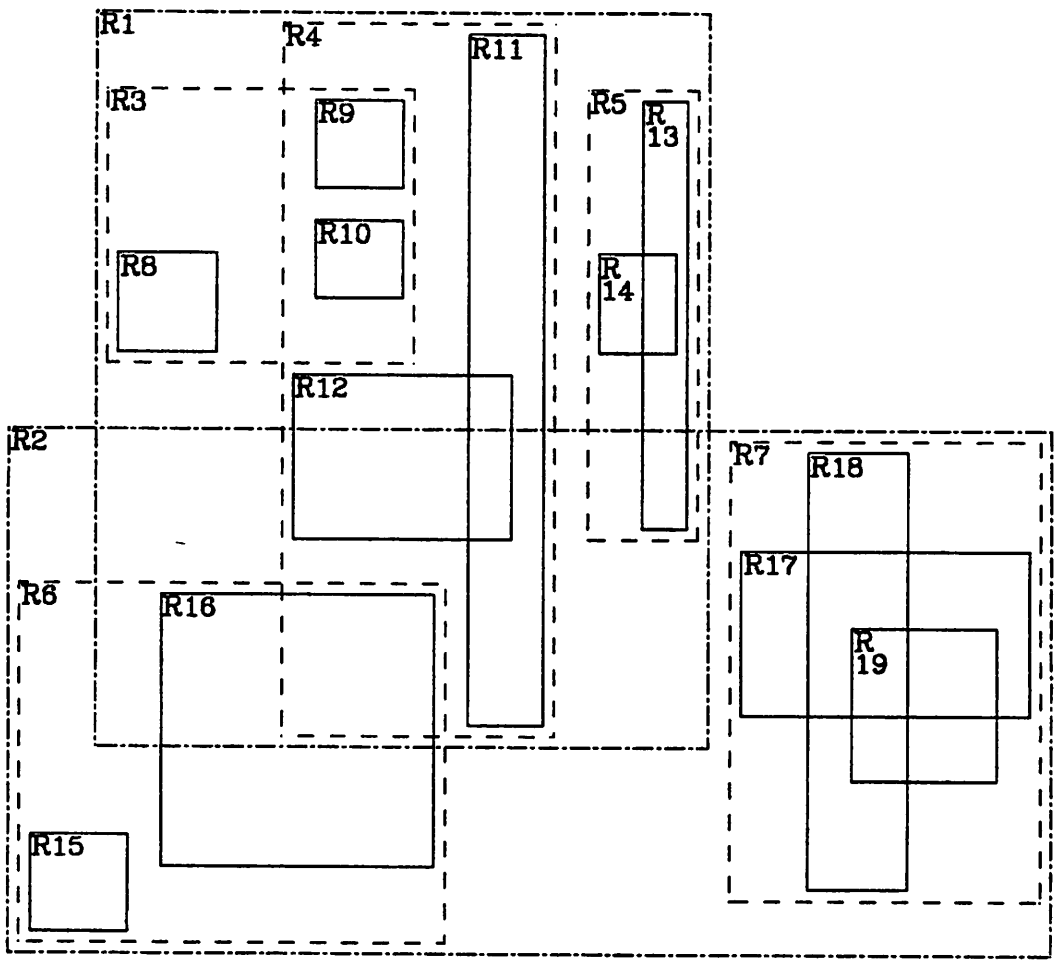
3. Searching and Updating

3.1. Searching

The search algorithm descends the tree from the root in a manner similar to a B-tree. However when it visits a non-leaf node it may find that any number of subtrees from 0 to M need to be searched; hence it is not possible to guarantee good worst-case performance. Nevertheless with most kinds of data the update algorithms will maintain the tree in a form that allows the search algorithm to eliminate irrelevant regions of the indexed space, and examine only data near the search area.


R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14
R15
R16
R17
R18
R19

(a)


R1
R4
R11
R3
R9
R5
R13
R10
R8
R14
R12
R2
R7
R18
R17
R6
R16
R19
R15

(b)
Figure 2.1

In the following we denote the rectangle part of an index entry E by $E.I$, and the *tuple-identifier* or *child-pointer* part by $E.p$.

Algorithm **Search.** Given an R-tree whose root node is T, find all index records whose rectangles overlap a search rectangle S.

S1. [Search subtrees.]

If T is not a leaf, check each entry E to determine whether $E.I$ overlaps S. For all overlapping entries, invoke **Search** on the tree whose root node is pointed to by $E.p$.

S2. [Search leaf node.]

If T is a leaf, check all entries E to determine whether $E.I$ overlaps S. If so, E is a qualifying record.

3.2. Insertion

Inserting index records for new data tuples is similar to insertion in a B-tree in that new index records are added to the leaves, nodes that overflow are split, and splits propagate up the tree.

Algorithm **Insert.** Insert a new index entry E into an R-tree.

I1. [Find position for new record.]

Invoke **ChooseLeaf** to select a leaf node L in which to place E.

I2. [Add record to leaf node.]

If L has room for another entry, install E. Otherwise invoke **SplitNode** to obtain L and LL containing E and all the old entries of L.

I3. [Propagate changes upward.]

Invoke **ExpandTree** on L, also passing LL if a split was performed.

I4. [Grow tree taller.]

If node split propagation resulted in the root being split, create a new root whose children are the two nodes resulting from the split.

Algorithm **ChooseLeaf**. Select a leaf node of an R-tree in which to place a new index entry E.

CL1. [Initialize.]

Set N to be the root node.

CL2. [Leaf check.]

If N is a leaf, return N.

CL3. [Choose subtree.]

If N is not a leaf, let F be the entry in N whose rectangle $F.I$ needs least enlargement to include $E.I$. Resolve ties by choosing the entry with the rectangle of smallest area.

CL4. [Descend until a leaf is reached.]

Set N to be the child node pointed to by $F.p$ and repeat from CL2.

Algorithm **ExpandTree**. Ascend from a leaf node L in an R-tree to the root, adjusting covering rectangles and propagating node splits as necessary.

ET1. [Initialize.]

Set $N{=}L$. If L was split previously, set NN to be the resulting second node.

ET2. [Check if done.]

If N is the root, stop.

ET3. [Adjust covering rectangle in parent entry.]

Let P be the parent node of N, and let E_N be N's entry in P. Adjust $E_N.I$ so that it tightly encloses all entry rectangles in N.

ET4. [Propagate node split upward.]

If N has a partner NN resulting from an earlier split, create a new entry E_{NN} with $E_{NN}.p$ pointing to NN and $E_{NN}.I$ enclosing all rectangles in NN. Add E_{NN} to P if there is room. Otherwise, invoke **SplitNode** to produce P and PP containing E_{NN} and all P's old entries.

ET5. [Move up to next level.]

Set $N=P$ and set $NN=PP$ if a split occurred. Repeat from ET2.

Algorithm **SplitNode** is described in Section 3.5.

3.3. Deletion

Algorithm **Delete**. Remove index record E from an R-tree.

D1. [Find node containing record.]

Invoke **FindLeaf** to locate the leaf node L containing E. Stop if the record was not found.

D2. [Delete record.]

Remove E from L.

D3. [Adjust tree.]

Invoke **CondenseTree** to adjust the covering rectangles on the path from L to the root, to eliminate under-full nodes, and to propagate node eliminations up the tree.

D4. [Shorten tree.]

If the root node has only one child after the tree has been adjusted, make the child the new root.

Algorithm **FindLeaf.** Given an R-tree whose root node is T, find the leaf node containing the index entry E.

FL1. [Search subtrees.]

If T is not a leaf, check each entry F in T to determine if $F.I$ overlaps $E.I$. For each such entry invoke **FindLeaf** on the tree whose root is pointed to by $F.p$ until E is found or all entries have been checked.

FL2. [Search leaf node for record.]

If T is a leaf, check each entry to see if it matches E. If E is found return T.

Algorithm **CondenseTree.** Given an R-tree leaf node L from which an entry has been deleted, eliminate the node if it has too few entries and relocate its entries. Propagate node elimination upward as necessary. Adjust all covering rectangles on the path to the root, making them smaller if possible.

CT1. [Initialize.]

Set $N = L$. Set Q, the set of eliminated nodes, to be empty.

CT2. [Find parent entry unless root has been reached.]

If N is the root, go to CT6. Otherwise let P be the parent of N, and let E_N be N's entry in P.

CT3. [Eliminate under-full node.]

If N has fewer than m entries, delete E_N from P and add N to set Q.

CT4. [Adjust covering rectangle.]

If N has not been eliminated, adjust $E_N.I$ to tightly contain all entries in N.

CT5. [Move up one level in tree.]

Set $N{=}P$ and repeat from CT2.

CT6. [Re-insert orphaned entries.]

Re-insert all entries of nodes in set Q. Entries from eliminated leaf nodes are re-inserted in tree leaves as described in Algorithm **Insert**, but entries from higher-level nodes must be placed higher in the tree. This is done so that leaves of their dependent subtrees will be on the same level as leaves of the main tree.

The procedure outlined above for disposing of under-full nodes differs from the corresponding operation on a B-tree, in which two or more adjacent nodes are merged. A B-tree-like approach is possible for R-trees, although there is no adjacency in the B-tree sense: an under-full node can be merged with whichever sibling will have its area increased least, or the orphaned entries can be distributed among sibling nodes. Either method can cause nodes to be split.

Re-insertion was chosen instead for two reasons: first, it accomplishes the same thing and is easier to implement because the **Insert** routine can be used. Efficiency should be comparable because pages needed during re-insertion usually will be the same ones visited during the preceding search and will already be in memory. The second reason is that re-insertion incrementally refines the spatial structure of the tree, and prevents gradual deterioration that might occur if each entry were located permanently under the same parent node.

3.4. Updates and Other Operations

If a data tuple is updated so that its covering rectangle is changed, its index record must be deleted, updated, and then re-inserted, so that it will find its way to the right place in the tree.

Other kinds of searches besides the one described above may be useful, for example to find all data objects completely contained in a search area, or all objects that contain a search area. These operations can be implemented by straightforward variations on the algorithm given. A search for a specific entry whose identity is known beforehand is required by the deletion algorithm and is implemented by Algorithm **FindLeaf**. Variants of range deletion, in which index entries for all data objects in a particular area are removed, are also well supported by R-trees.

3.5. Node Splitting

When an attempt is made to add an entry to a full node containing M entries, the collection of $M+1$ entries must be divided between two nodes. The division should be done in a way that makes it as unlikely as possible that both new nodes will need to be examined on subsequent searches. Since the decision to visit a node is based on whether the search area overlaps the covering rectangle for the node's entries, the total area of the two covering rectangles should be minimized. Figure 3.1 illustrates this point. The area of the covering rectangles in the "bad split" case is much larger than in the "good split" case.

Note that the same criterion was used in procedure ChooseLeaf to decide where to insert a new index entry: at each level in the tree, the subtree was chosen whose covering rectangle would have to be enlarged least.

We now turn to algorithms for partitioning the set of $M+1$ entries into two groups, one for each new page.

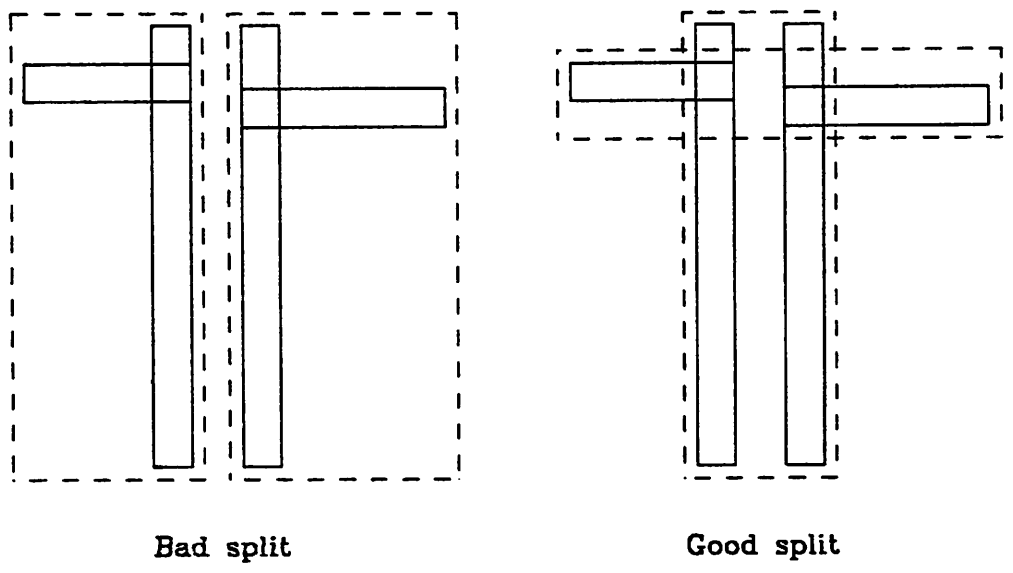


Figure 3.1

3.5.1. Exhaustive Algorithm

The most straightforward way to find the minimum area node split is to generate all alternatives and choose the best. However, the number of possible partitions is approximately 2^{M-1} and a reasonable value of M is 50*, so the number of possible splits is very large. We implemented a modified form of the exhaustive node split algorithm to use as a standard for comparison with other algorithms, but it was too slow to use with large page sizes.

3.5.2. A Quadratic-Cost Algorithm

This algorithm attempts to find a small-area split, but is not guaranteed to find one with the smallest area possible. The cost is quadratic in M and linear in the number of dimensions. The algorithm picks two of the $M+1$ entries to be the first elements of the groups which will make up the new nodes. The pair chosen is the one that would waste the most area if both entries were put in the same group, i.e. the area of the rectangle covering both entries, minus the areas of the rectangles in the entries, would be greatest. Then the remaining entries are selected one at a time and assigned to a group. At each step the area expansion required to add each entry to each group is calculated, and the entry chosen is the one showing the greatest difference between the two groups in the expansion required to include it.

Algorithm **Quadratic Split**. Divide a set of $M+1$ index entries into two groups.

* A two dimensional rectangle can be represented by four numbers of four bytes each. If a pointer also takes four bytes, each entry requires 20 bytes. A page of 1024 bytes will hold about 50 entries.

QS1. [Pick first entry for each group.]

Apply Algorithm **PickSeeds** to choose two entries to be the first elements of the groups. Assign each to a group.

QS2. [Check if done.]

If all entries have been assigned, stop. If one group has so few entries that all the rest must be assigned to it in order for it to have the minimum number m, assign them and stop.

QS3. [Select entry to assign.]

Invoke Algorithm **PickNext** to choose the next entry to assign. Add it to the group whose covering rectangle will have to be enlarged least to accommodate it. Resolve ties by adding the entry to the group with smaller area, then to the one with fewer entries, then to either. Repeat from QS2.

Algorithm **PickSeeds**. Select two entries to be the first elements of the groups.

PS1. [Calculate inefficiency of grouping entries together.]

For each pair of entries E_1 and E_2, compose a rectangle J including $E_1.I$ and $E_2.I$. Calculate $d=$ area(J) - area($E_1.I$) - area($E_2.I$).

PS2. [Choose the most wasteful pair.]

Choose the pair with the largest d to be put in different groups.

Algorithm **PickNext**. Select one remaining entry for classification in a group.

PN1. [Determine cost of putting each entry in each group.]

For each entry E not yet in a group, calculate $d_1=$ the area increase required in the covering rectangle of Group 1 to include $E.I$. Calculate d_2 similarly for

Group 2.

PN2. [Find entry with greatest preference for one group.]

Choose the entry with the greatest difference between d_1 and d_2. If more than one has the same lowest difference pick any of them.

3.5.3. A Linear-Cost Algorithm

This algorithm is linear in M and in the number of dimensions. First it selects seed entries for the two groups, choosing the two whose rectangles are the most widely separated along any dimension. Then it processes the remaining entries without ordering them in any special way, placing each in one of the groups.

Algorithm **Linear Split** is identical to **Quadratic Split** but uses a different version of **PickSeeds**. **PickNext** simply chooses any of the remaining entries.

Algorithm **LinearPickSeeds**. Select two entries to be the first elements of the groups.

LPS1. [Find extreme rectangles along all dimensions.]

Along each dimension, find the entry whose rectangle has the highest low side, and the one with the lowest high side. Record the separation.

LPS2. [Adjust for shape of the rectangle cluster.]

Normalize the separations by dividing by the width of the entire set along the corresponding dimension.

LPS3. [Select the most extreme pair.]

Choose the pair with the greatest normalized separation along any dimension.

4. Performance Tests

We implemented R-trees in C under Unix on a Vax 11/780 computer. Our implementation has been used for a series of performance tests, whose purpose was to verify the practicality of the structure, to choose values for M and m, and to evaluate different node-splitting algorithms. This section presents the results.

Five page sizes were tested, corresponding to different values of M:

Bytes per Page	Max Entries per Page (M)
128	6
256	12
512	25
1024	50
2048	102

The minimum number of entries in a node (m) was tested for values $M/2$, $M/3$, and 2. The three node split algorithms described earlier were implemented in different versions of the program.

All our tests used two-dimensional data, although the structure and algorithms work for any number of dimensions. During the first part of each test run the program read geometry data from files and constructed an index tree, beginning with an empty tree and calling *Insert* with each new index record. Insert performance was measured for the last 10% of the records, when the tree was nearly its final size.

During the second phase the program called the function *Search* with random search rectangles made up using the local random number generation facility. 100 searches were performed per test run, each retrieving about 5% of the data.

Finally the program read the input files a second time and called the function *Delete* to remove the index record for every tenth data item. Thus measurements were taken for scattered deletion of 10% of the index records.

The tests were done using Very Large Scale Integrated circuit (VLSI) layout data from the RISC-II computer chip. [11]. The first series of tests used the circuit cell CENTRAL, containing 1057 rectangles (Figure 4.1).

Figure 4.2 shows the cost in CPU time for inserting the last 10% of the records as a function of page size. The exhaustive algorithm, whose cost increases exponentially with page size, is seen to be very slow for larger page sizes. The linear algorithm is fastest, as expected. With this algorithm CPU time hardly increased with page size at all, which suggests that node splitting was responsible for only a small

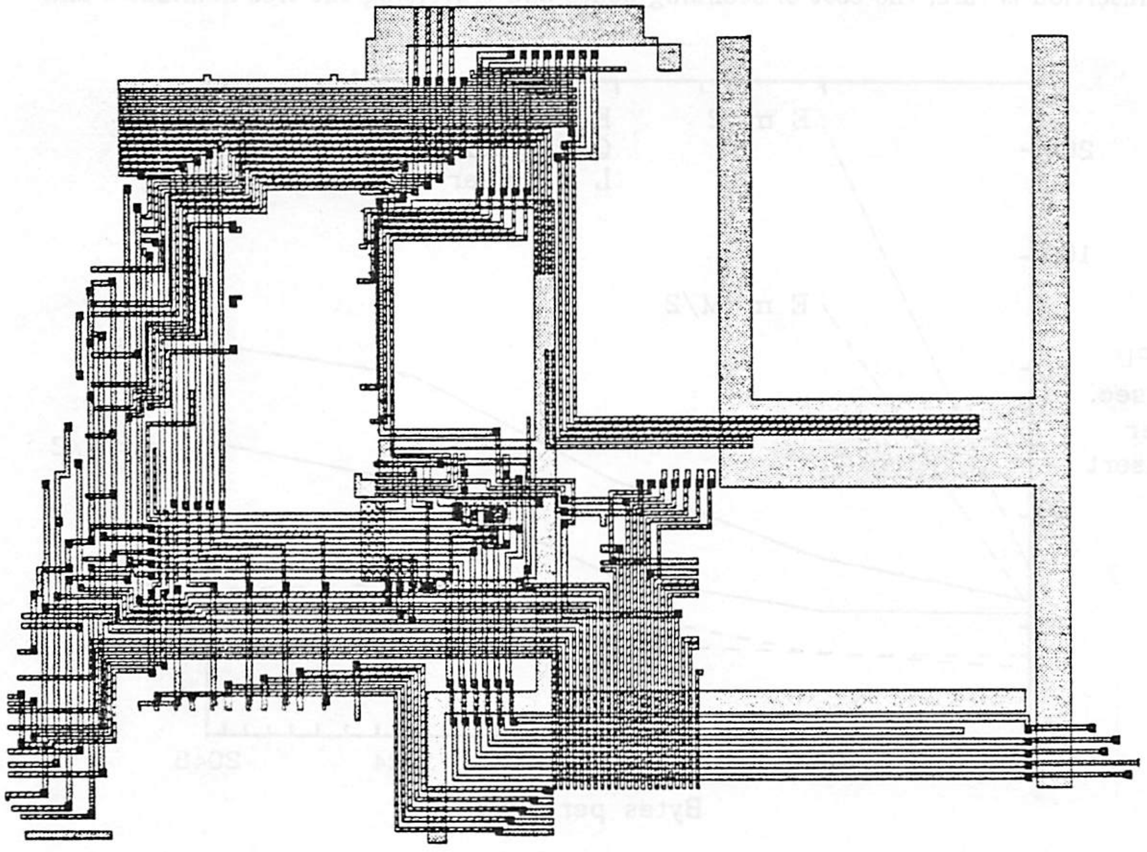


Figure 4.1
Circuit cell CENTRAL (1057 rectangles).

part of the cost of inserting records.

The decreased cost of the quadratic algorithm with a stricter node fill requirement reflects the fact that it stops doing expensive processing when one group becomes too full, and changes to a linear mode for the rest of the split.

The cost of deleting an item from the index, shown in Figure 4.3, is affected by the split algorithm and especially by the minimum node fill requirement. This is true because deletion indirectly causes node splitting when entries from under-full nodes are re-inserted. Stricter fill requirements cause nodes to be eliminated more frequently, and each one has more entries to re-insert. When the minimum fill is 2 re-insertion is rare; the cost of scanning nodes and traversing the tree dominates and

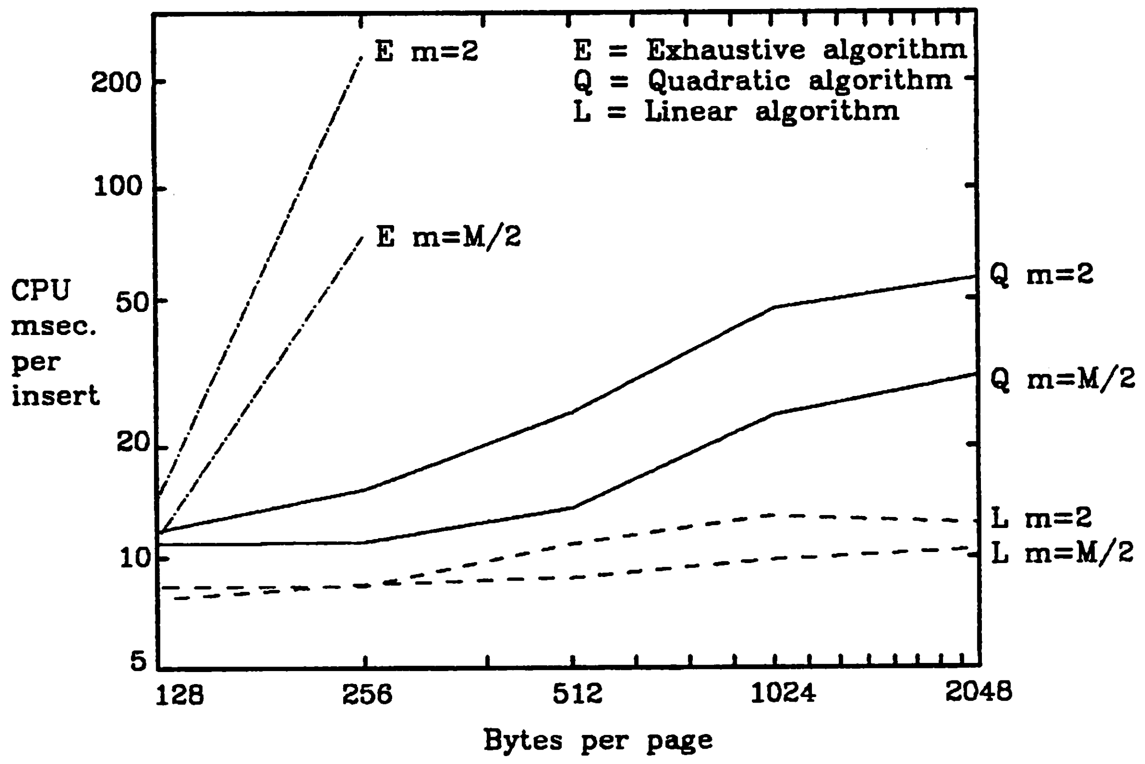


Figure 4.2
CPU cost of inserting records.

the split algorithm has little effect. The curves are rough because node eliminations occur randomly and infrequently; there were too few of them in our tests to smooth out the variations.

Figures 4.4 and 4.5 show that the search performance of the index is very insensitive to the use of different node split algorithms and fill requirements. The exhaustive algorithm produces a slightly better index structure, resulting in fewer pages touched and less CPU cost, but even the crudest algorithm, the linear one with $m = M/2$, provides reasonable performance. Most combinations of algorithm and minimum fill come within 10% of the performance achieved with node splits produced by the exhaustive algorithm.

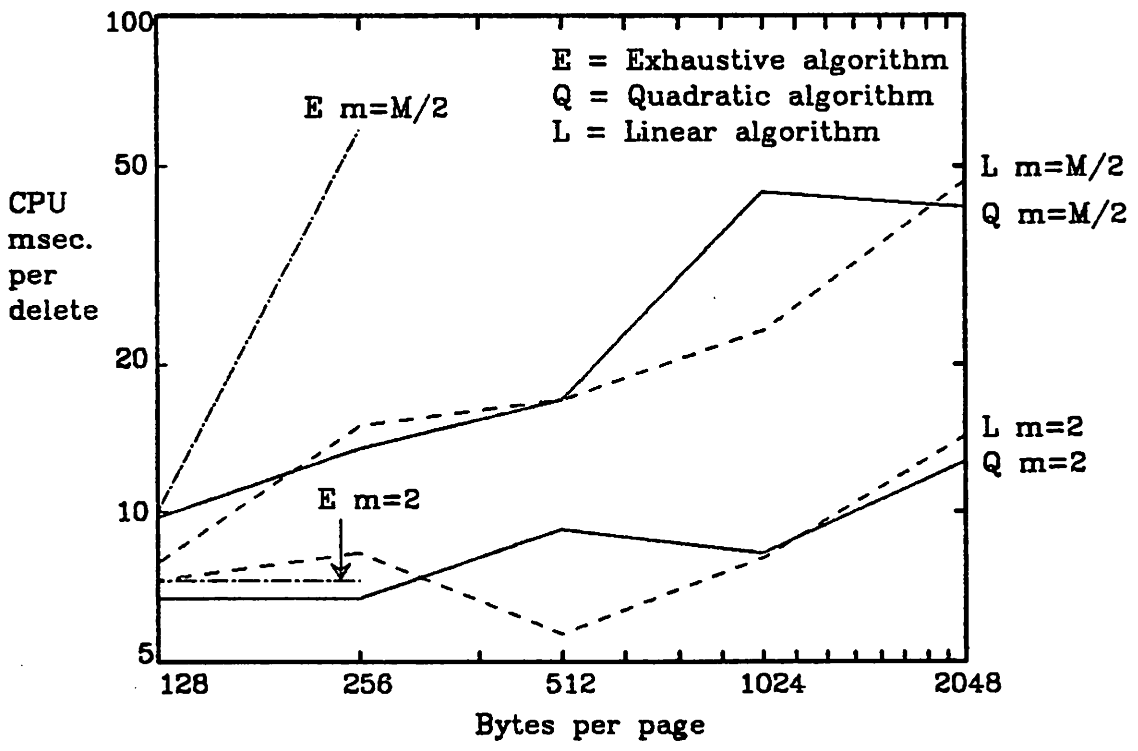


Figure 4.3
CPU cost of deleting records.

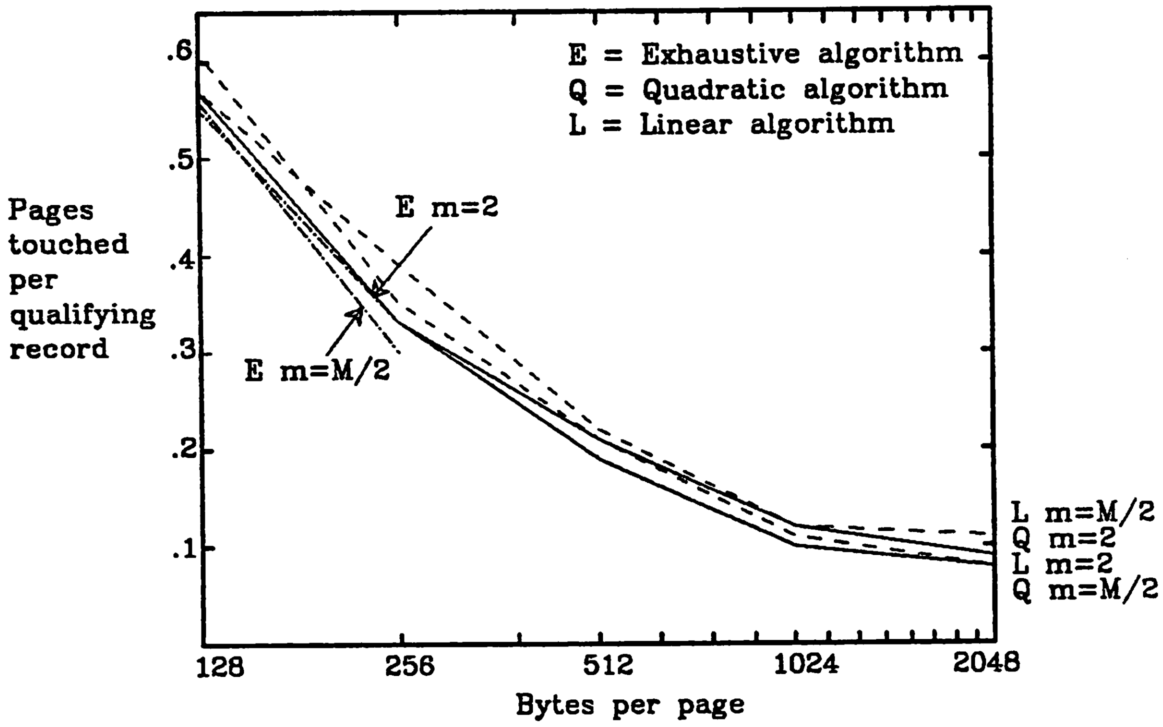


Figure 4.4
Search performance: Pages touched.

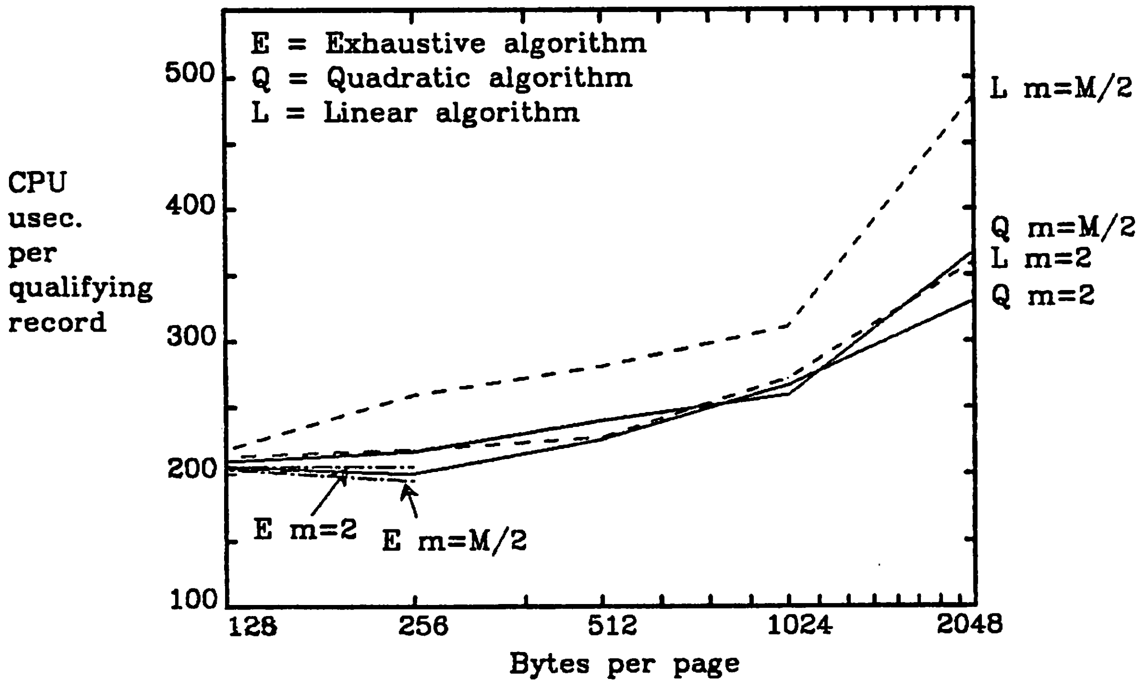


Figure 4.5
Search performance: CPU cost.

Figure 4.6 shows the storage space occupied by the index tree as a function of algorithm, fill criterion and page size. Generally the results bear out our expectation that stricter node fill criteria produce smaller indexes. The least dense index consumes about 50% more space than the most dense, but all results for 1/2-full and 1/3-full (not shown) are within 15% of each other.

We performed a second series of tests to measure R-tree performance as a function of the amount of data in the index. The same sequence of test operations as before was run on samples containing 1057, 2238, 3295, and 4559 rectangles. The first sample contained layout data from the same circuit cell CENTRAL as used earlier. The second sample consisted of layout from a larger circuit cell containing 2238 rectangles. The third sample was made by using both CENTRAL and the larger

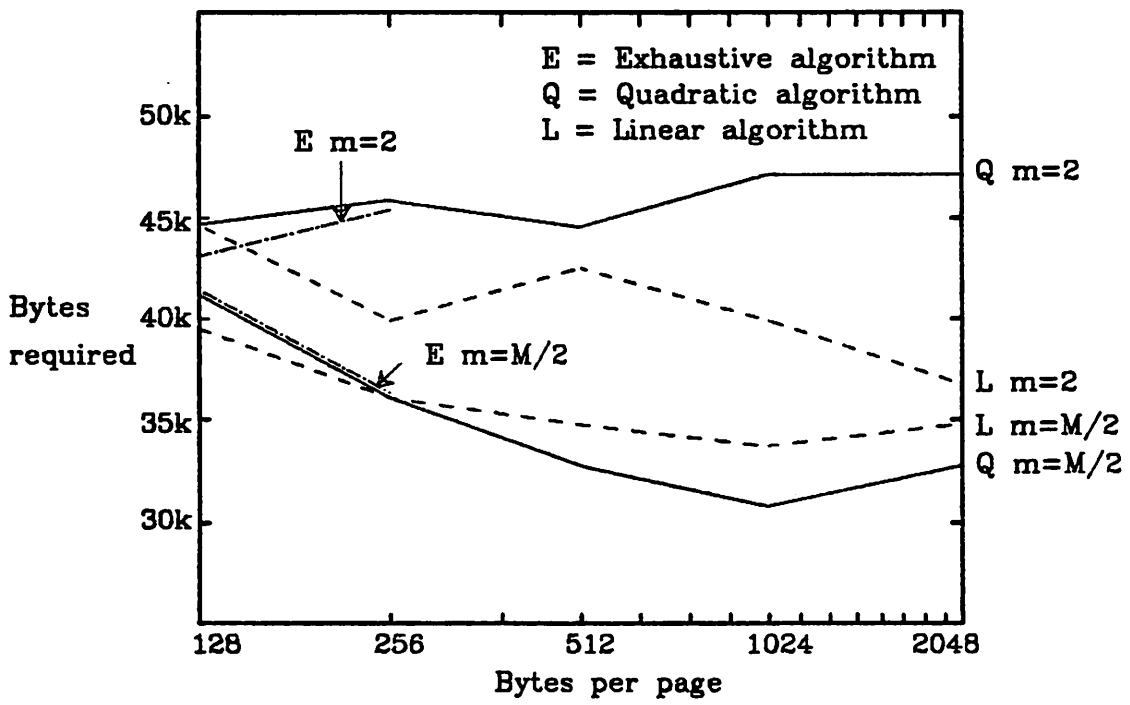


Figure 4.6
Space efficiency.

cell, with the two cells effectively placed on top of each other. Three cells were combined to make up the last sample. Because the samples were all composed in different ways using varying data, performance results would not scale perfectly and some unevenness was to be expected.

Two combinations of algorithm and node fill requirement were chosen for the tests: the linear split algorithm with m=2, and the quadratic algorithm with m=M/3, both with a page size of 1024 bytes (M=50). Both have good search performance; the linear configuration has about half the insert cost, and the quadratic one produces a smaller index.

Figure 4.7 shows the results of tests whose purpose was to determine how insert and delete performance is affected by tree size. Both test configurations produced trees with two levels for 1057 records and three levels for the other sample sizes. The figure shows that the cost of inserts with the quadratic algorithm is nearly constant except where the tree increases in height, where the curve shows a definite jump because the number of levels where an expensive split can occur increases. The linear algorithm shows no jump, indicating again that linear node splits account for only a small part of the cost of inserts.

Deletion with the linear configuration has nearly constant cost for fixed tree height, and a small jump where the tree becomes taller. The tests involved too few deletions to cause any node splits on re-insertion with the relaxed node fill requirement. A few splits did occur during deletion with the quadratic configuration, but only 1 to 6 per test run, and the number of nodes eliminated was similarly small, varying between 4 and 15. Such small numbers, coupled with the high cost of node

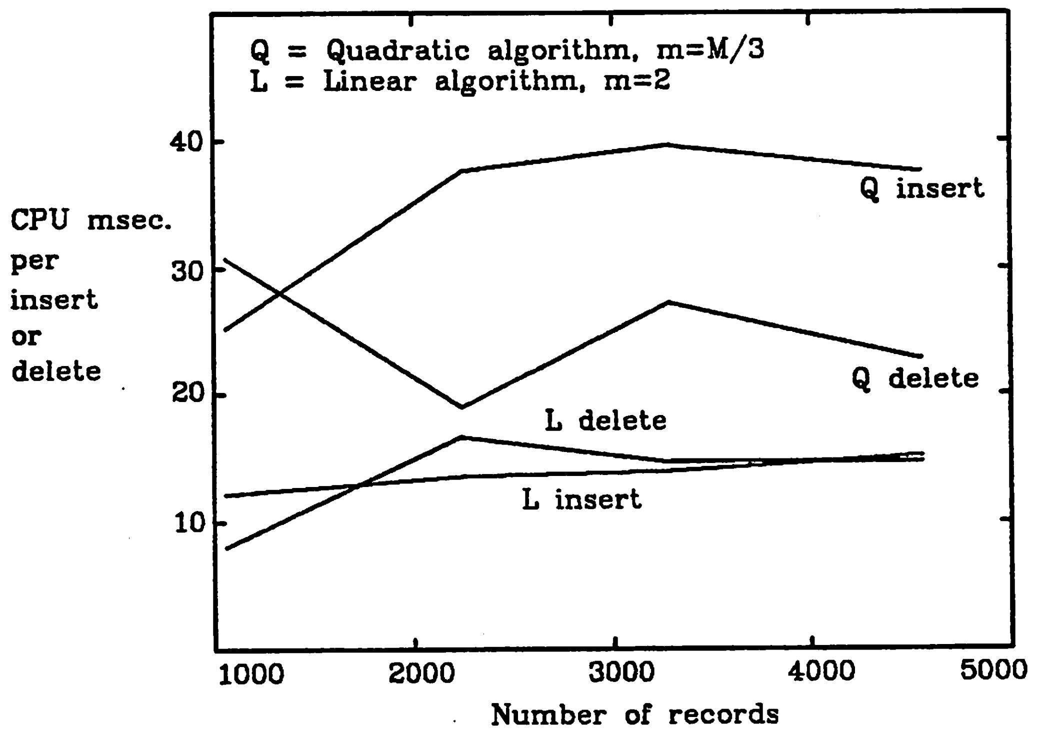


Figure 4.7
CPU cost of inserts and deletes vs. amount of data.

eliminations, made the quadratic curve very erratic.

When allowance is made for variations due to the small sample size, the tests show that insert and delete cost is independent of tree width but is affected by tree height, which grows slowly with the number of data items. This performance is very similar to that of a B-tree.

The search tests retrieved between 3% and 6% of the data on each search. Figures 4.8 and 4.9 confirm that the two configurations have nearly the same performance. The downward trend of the curves is to be expected, because the cost of processing higher tree nodes becomes less significant as the amount of data retrieved in each search increases. The decrease would have been more uniform, but an unexpected variation in the 2238-item sample raised the cost at the second data point.

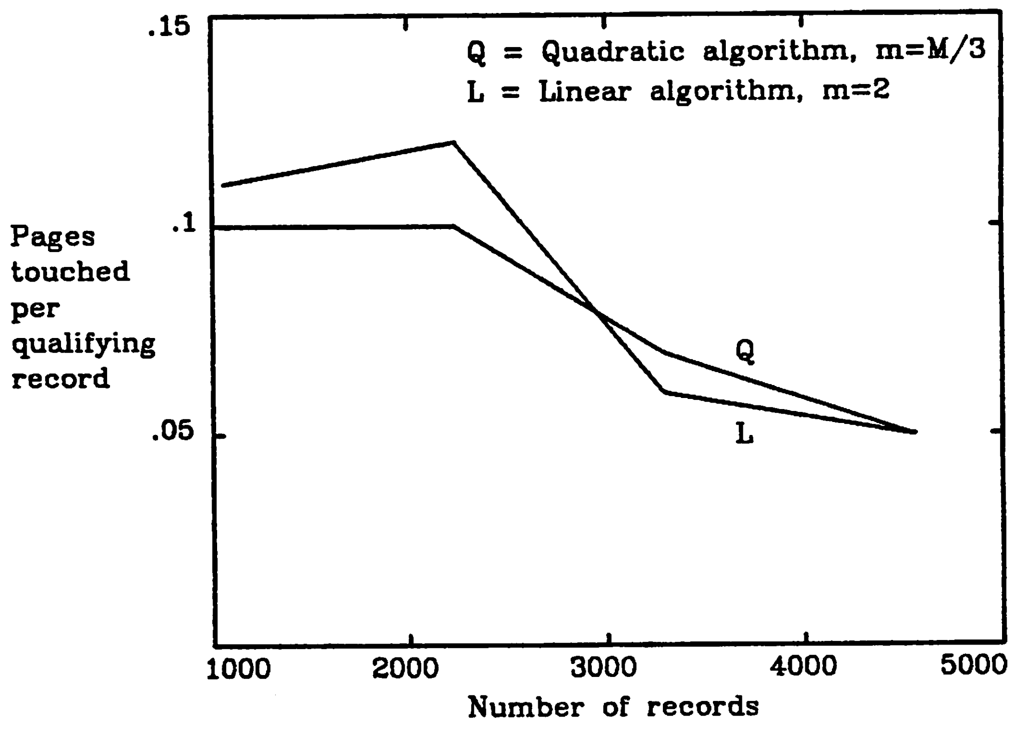


Figure 4.8
Search performance vs. amount of data: Pages touched

This sample was considerably denser than the others and contained more long shapes spanning many small ones. The search rectangles tended to overlap a larger fraction of the bins in the index, which resulted in less efficient searching.

The low CPU cost per qualifying record, less than 150 microseconds for larger amounts of data, shows that the index is quite effective in narrowing searches to small subtrees.

The straight lines in Figure 4.10 reflect the fact that almost all the space in an R-tree index is used for leaf nodes, whose number varies linearly with the amount of data. For the linear-2 test configuration the total space occupied by the R-tree was about 40 bytes per data item, compared to 20 bytes per item for the index records alone. The corresponding figure for the quadratic-1/3 configuration was about 33

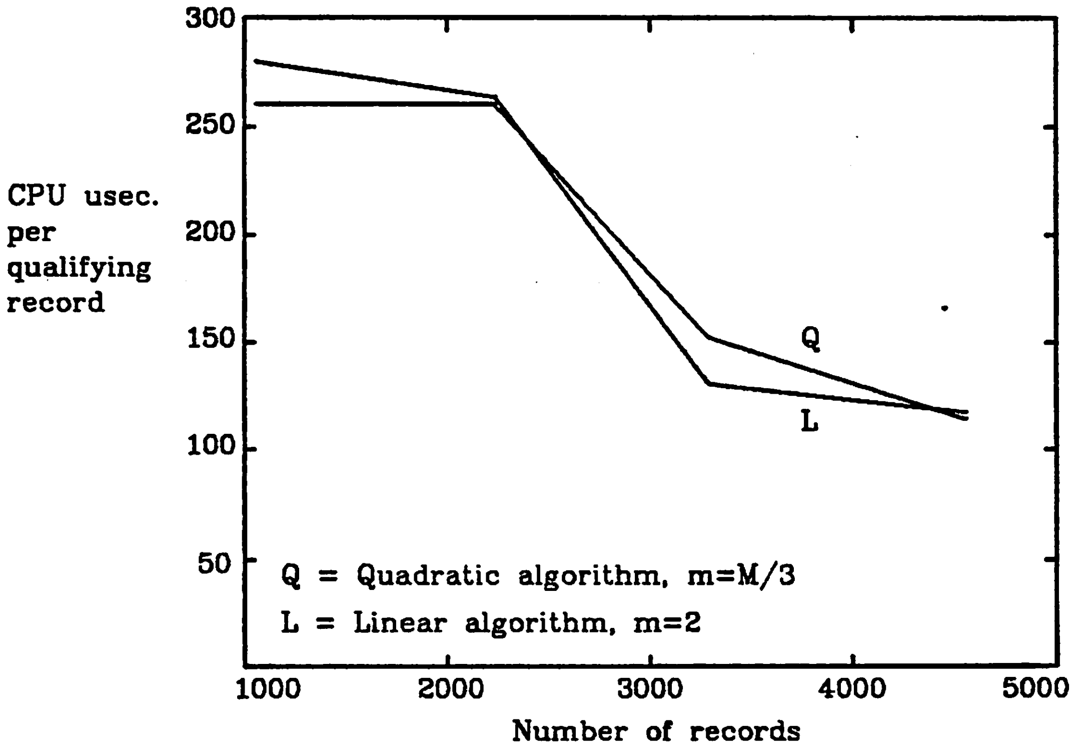


Figure 4.9
Search performance vs. amount of data: CPU cost

bytes per item.

5. Conclusions

The R-tree structure has been shown to be useful for indexing spatial data objects that have non-zero size. Nodes corresponding to disk pages of reasonable size (e.g. 1024 bytes) have values of M that produce good performance. With smaller nodes the structure should also be effective as a main-memory index; CPU performance would be comparable but there would be no I/O cost.

The linear node-split algorithm proved to be as good as more expensive techniques. It was fast, and the slightly worse quality of the splits did not affect search performance noticeably.

Preliminary investigation indicates that R-trees would be easy to add to any relational database system that supported conventional access methods, (e.g. INGRES [9], System-R [1]). Moreover, the new structure would work especially well in conjunction with abstract data types and abstract indexes [14] to streamline the handling of spatial data.

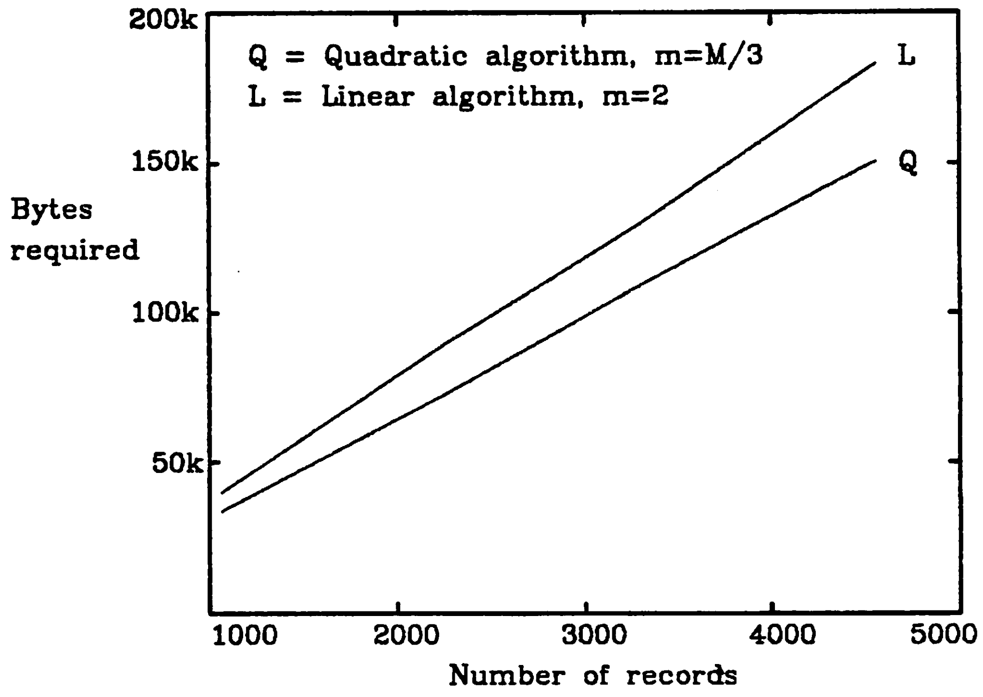


Figure 5.10
Space required for R-tree vs. amount of data.

REFERENCES

[1] M. Astrahan, et al., "System R: Relational Approach to Database Management," *ACM Transactions on Database Systems* 1(2) pp. 97-137 (June 1976).

[2] R. Bayer and E. McCreight, "Organization and Maintenance of Large Ordered Indices," *Proc. 1970 ACM-SIGFIDET Workshop on Data Description and Access*, pp. 107-141 (Nov. 1970).

[3] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM* 18(9) pp. 509-517 (September 1975).

[4] J. L. Bentley, D. F. Stanat, and E. H. Williams Jr., "The complexity of fixed-radius near neighbor searching," *Inf. Proc. Lett.* 6(6) pp. 209-212 (December 1977).

[5] J. L. Bentley and J. H. Friedman, "Data Structures for Range Searching," *Computing Surveys* 11(4) pp. 397-409 (December 1979).

[6] D. Comer, "The Ubiquitous B-tree," *Computing Surveys* 11(2) pp. 121-138 (1979).

[7] R. A. Finkel and J. L. Bentley, "Quad Trees - A Data Structure for Retrieval on Composite Keys," *Acta Informatica* 4 pp. 1-9 (1974).

[8] A. Guttman and M. Stonebraker, "Using a Relational Database Management System for Computer Aided Design Data," *IEEE Database Engineering* 5(2)(June 1982).

[9] G. Held, M. Stonebraker, and E. Wong, "INGRES - A Relational Data Base System," *Proc. AFIPS 1975 NCC* 44 pp. 409-416 (1975).

[10] K. Hinrichs and J. Nievergelt, "The Grid File: A Data Structure Designed to Support Proximity Queries on Spatial Objects," Nr. 54, Institut fur Informatik, Eidgenossische Technische Hochschule, Zurich (July 1983).

[11] M.G.H. Katevenis, R.W. Sherburne, D.A. Patterson, and C.H. Séquin, "The RISC II Micro-Architecture," *Proc. VLSI 83 Conference*, (August 1983).

[12] J. K. Ousterhout, "Corner Stitching: A Data Structuring Technique for VLSI Layout Tools," Computer Science Report CSD 82/114, University of California, Berkeley (1982).

[13] J. T. Robinson, "The K-D-B Tree: A Search Structure for Large Multidimensional Dynamic Indexes," *ACM-SIGMOD Conference Proc.*, pp. 10-18 (April 1981).

[14] M. Stonebraker, B. Rubenstein, and A. Guttman, "Application of Abstract Data Types and Abstract Indices to CAD Data Bases," Memorandum No. UCB/ERL M83/3, Electronics Research Laboratory, University of California, Berkeley (January 1983).

[15] K. C. Wong and M. Edelberg, "Interval Hierarchies and Their Application to

Predicate Files," *ACM Transactions on Database Systems* **2**(3) pp. 223-232 (September 1977).

[16] G. Yuval, "Finding Near Neighbors in k-dimensional Space," *Inf. Proc. Lett.* **3**(4) pp. 113-114 (March 1975).