S-TREES: DATABASE INDEXING TECHNIQUES FOR MULTI-DIMENSIONAL INTERVAL DATA

by

Curtis P. Kolovson and Michael Stonebraker

Memorandum No. UCB/ERL M90/35

27 April 1990

S-TREES: DATABASE INDEXING TECHNIQUES FOR MULTI-DIMENSIONAL INTERVAL DATA

by

Curtis P. Kolovson and Michael Stonebraker

Memorandum No. UCB/ERL M90/35

27 April 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

S-TREES: DATABASE INDEXING TECHNIQUES FOR MULTI-DIMENSIONAL INTERVAL DATA

by

Curtis P. Kolovson and Michael Stonebraker

Memorandum No. UCB/ERL M90/35

27 April 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

S-TREES: DATABASE INDEXING TECHNIQUES FOR MULTI-DIMENSIONAL INTERVAL DATA

Curtis P. Kolovson and Michael Stonebraker

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

Abstract

We propose new techniques for indexing interval data in K dimensions, K >= 1, which are based on a set of extensions to existing database indexing structures. These extensions may be applied to multi-attribute spatial data indexing structures such as R-Trees, as well as to one-dimensional indexes such as B-Trees. We present the extensions, the motivation for these techniques, describe how they would be applied to R-Trees and B-Trees, and present the results of a performance study which demonstrates the soundness of our approach.

1. Introduction

Interval data is widely used in spatial and geometric applications, as well as for representing *historical* (temporal) data. Such data may be characterized by a set of ordered pairs that specify lower and upper bounds in K dimensions, K >= 1. Currently, several proposals have been made for data structures that support efficient searching of large collections of multi-dimensional interval data. We broadly categorize these proposals into two significant classes:

(1) Main memory based data structures used in Computational Geometry [PREP85], and

(2) Disk based indexing structures used in Database Management Systems [SAME89].

The data structures that have been developed in the field of Computational Geometry for indexing interval data are based on variations of binary search trees. These include the Segment Tree [BENT77], Interval Tree [EDEL80], Priority Search Tree [MCCR85], and Persistent Search Tree [SARN86]. All of these data structures are binary tree structures that were designed with the assumption that the entire struc-

---

This research was sponsored by the Army Research Organization Grant DAAL03-87-0083 and by the Defense Advanced Research Projects Agency through NASA Grant NAG2-530

ture is contained in main memory, and none have been extended to n-ary trees for an environment where the structure is paged onto secondary storage. The method of Interval Hierarchies [WONG77] cannot be used in multiple dimensions and is also a main memory based data structure that has not been extended to work in a paged environment.

There are several indexing techniques proposed for database management systems for various forms of interval and historical data. The Write-Once B-Tree [EAST86, LOME89] is potentially wasteful of secondary storage space since large portions of the index may contain redundant information that has been replicated multiple times on disk. The R-Tree [GUTT84] and R+-Tree [SELL87] store all interval data in the leaf nodes. Neither of these indexing techniques are particularly well-suited to dealing with interval data whose length distribution is highly non-uniform. In particular, neither performs well given a high proportion of "short" intervals and a low proportion of "long" intervals. There are likely to be many collections of interval data whose length distributions are non-uniform. For example, Figure 1 illustrates historical data representing employee salary histories, where the horizontal X-axis represents *time* and the vertical Y-axis represents employee salaries. In this figure, a collection of historical data is represented by a set of horizontal line segments in two dimensions which are parallel to the *year* axis. Such a data collection is likely to consist of mostly short intervals corresponding to employees who received frequent salary raises, and a small proportion of very long intervals (employees who seldom received raises).

In this paper, we combine aspects of the data structures from (1) and (2) above to provide efficient indexing techniques for multi-dimensional interval data in a database environment where only a small portion of the index may reside in main memory at any given time. We name our indexing structures *S-Trees* to reflect their utility in searching collections of line *segment* (interval) data.

The remainder of this paper proceeds as follows. Section 2 presents our tactics and the motivation for our research. Section 3 describes two example S-Trees: one based on the R-Tree (*SR-Tree*), and the other based on the B-Tree [BAYE72] (*SB-Tree*). Section 4 discusses the results of experiments that compare the performance of the SR-Tree to the R-Tree. Section 5 provides an example of one of the S-Tree tactics, which we refer to as the *Lop-Sided* index. Section 6 presents a summary and our conclusions.

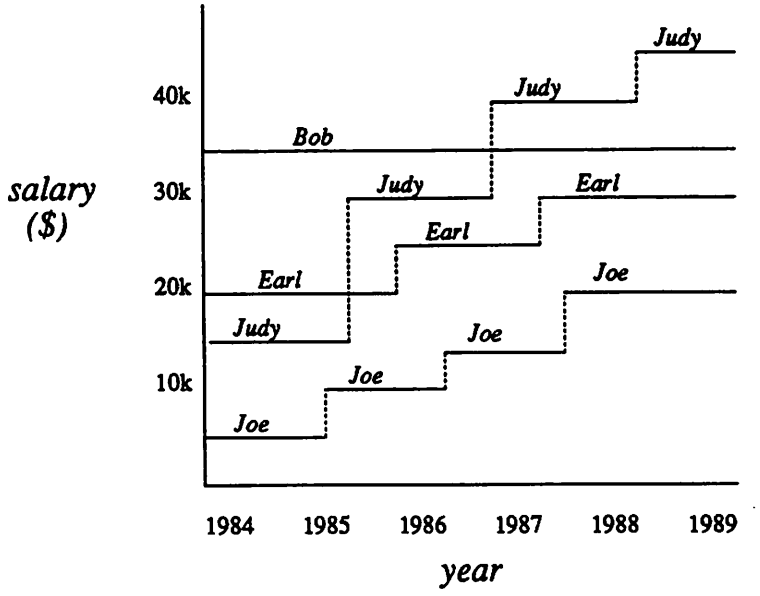


Figure 1: Historical data: Employee salaries as a function of time

2. The S-Tree Approach

When we speak of the S-Tree, we are not describing a specific indexing structure, but rather a number of tactics that may be applied to a class of tree-structured database indexing methods to improve their performance when dealing with interval data. Hereafter, we will prepend the letter *S* to the name of an indexing method that utilizes these tactics, as in *SB-Tree*. In this section, we present the tactics utilized in the various S-Tree indexes, and the motivation for S-Tree structures.

2.1. Tactics

This research investigates three major modifications to existing database indexing structures to support efficient search operations on multi-dimensional interval data:

(1) The index data may be stored at non-leaf nodes.

(2) The index page size may vary.

(3) Tree structures may become *Lop-Sided*, i.e., the balanced-tree criterion may be relaxed.

2.1.1. Storing Data in Non-Leaf Nodes

The first tactic is that an interval *I* shall be stored in the highest level node in a tree-structured index such that *I* spans all of the nodes that are descendants of one or more of the node's branches. In essence, we are extending Segment Trees from binary to n-ary trees.

2.1.2. Varying the Index Page Size

To support the first tactic, it may be desirable to have larger page sizes at successively higher levels in a tree-structured index. Since *external* index records (pointers to data records) and *internal* node branches (pointers to other index nodes) share space on a non-leaf index node in S-Tree structures, a non-leaf node with a large number of *external* index records will have a reduced branching factor. In order to maintain high fanout in such a tree-structured index, it is desirable to increase the size of a node at each successively higher level of the index.

2.1.3. Lop-Sided Trees

The motivation for *Lop-Sided trees* is to support a distribution of queries on interval data that is non-uniform in a particular dimension. For example, suppose that interval data is used to represent historical data, as in Figure 1. Given such a collection of data, queries involving recent historical data may occur more frequently than queries on older data. The assumption of a query distribution that is uniform over a particular attribute domain underlies the decision to maintain balanced tree-structured indexes. Clearly, a non-uniform query distribution over the time domain dictates a relaxation of the balance criterion of a tree-structured index. Therefore, our third tactic deals with modifying indexes to allow trees to become unbalanced based on a tunable *balance information* parameter that reflects the expected distribution of historical queries in the time domain.

2.2. Motivation for S-Trees

There are three motivating goals for this research, as follows:

(1) improve the performance of spatial indexing structures such as R-Trees,

(2) provide an efficient indexing technique for historical data using extended spatial indexing structures (a special case of (1)), and

(3) efficiently index one-dimensional line segments and point data in B-Trees.

The first motivation, to improve the performance of spatial indexing structures, arises from the observation that all tree-structured spatial indexing techniques store data items in the leaf nodes. We suggest that for certain input data distributions, such as one in which there is a high proportion of "small" objects and a low proportion of "large" objects, storing some of the larger objects in non-leaf nodes would result in a reduction in the amount of node coverage and/or overlap, thus improving the search performance of the index.

The second motivation is to provide indexing techniques for historical data [STON87], which may be characterized by an interval in the *time* dimension and a point in one or more other dimensions. Historical data values may be represented by a step function, as in Figure 1, where the location of a step corresponds to the commit-time of a transaction that changed an attribute value of a relation. One of our goals is to find efficient indexing techniques for such sets of interval data to support queries on historical data.

The third motivation is to efficiently index both line segment and point data in a B-Tree. This need may arise in database systems that support *rules*, such as POSTGRES [STON86]. For example, in a database that supports rules, suppose there is a currently active rule specifying that everyone who earns between $10K and $20K must have a wooden desk, and a new employee tuple is inserted into the Employee relation with a salary of $15K. In order to determine what sort of desk the new employee should be given, the appropriate rule must be applied. In general, whenever a new tuple is inserted, all of the rules which apply to the new tuple must be found and applied. In a geometric sense, the search for all the rules which apply to a newly inserted tuple corresponds to finding all of the intervals (associated with rules) that contain a specified point (an attribute value of a newly inserted tuple).

The next section of this paper presents two S-Trees: one based on the R-Tree and one based on the B-Tree. The R-Tree based scheme can be used for spatial or historical indexing while the B-Tree scheme supports intervals and points in the same structure.

3. Two Example S-Tree Indexes

The basic idea of S-Tree indexes is that data meeting certain criteria may be stored in the higher level nodes of the index. The manner in which this may be applied to a particular structure depends both

on the base indexing structure and the type of the data being indexed. In this section, we describe the design of two S-Tree indexes as defined in the previous section by describing the required modifications to the algorithms for insertion, node-splitting, and search operations. The first such index is based on the R-Tree for indexing multi-dimensional interval data, and the other is based on the B-Tree for indexing one-dimensional interval and point data.

3.1. SR-Tree

We define the SR-Tree as the S-Tree adaptation of the R-Tree index. The R-Tree is a K-dimensional variation of the B-Tree which indexes data consisting of intervals in K dimensions. With no loss of generality we will present a two-dimensional SR-Tree, as extensions to higher dimensions are straightforward.

3.1.1. Insert Algorithm

When a rectangle R is to be inserted, the algorithm descends the index beginning from the root. At each node, the branch B that would require the least expansion to accommodate R is selected, as in the original R-Tree algorithm. If the horizontal and/or vertical interval(s) of R span the corresponding dimension(s) of B, R is inserted onto this node and is added to a linked list of spanning objects associated with B, and the insertion algorithm does not descend the index below this node. If R does not span branch B in either dimension, the algorithm descends the index following branch B and applies the above algorithm. Hereafter, we refer to a data item stored on a non-leaf node as a *spanning rectangle* in the case of a two-dimensional SR-Tree, or more generally, as a *spanning object.*

A non-leaf node containing a spanning rectangle in a two-dimensional SR-Tree is illustrated in Figure 2. In this figure, two nodes are shown, labeled A and B. Node A contains a branch entry, E1, that stores both the spatial coordinates and disk address of node B. Suppose a newly inserted rectangle *R1* spans node B in the horizontal dimension but does not span node A in either dimension. *R1* is a spanning rectangle (it spans node B), and therefore an entry E2 is stored in node A which contains both the spatial coordinates of *R1* and the tuple ID for the database record corresponding to *R1*, and E2 is added to a linked list anchored at E1 of rectangles which span node B in one or both dimensions.

As in the original R-Tree, each rectangle inserted onto a leaf node must be enclosed by the region associated with the leaf node. Hence, if a newly inserted rectangle R is inserted onto a leaf node, it may

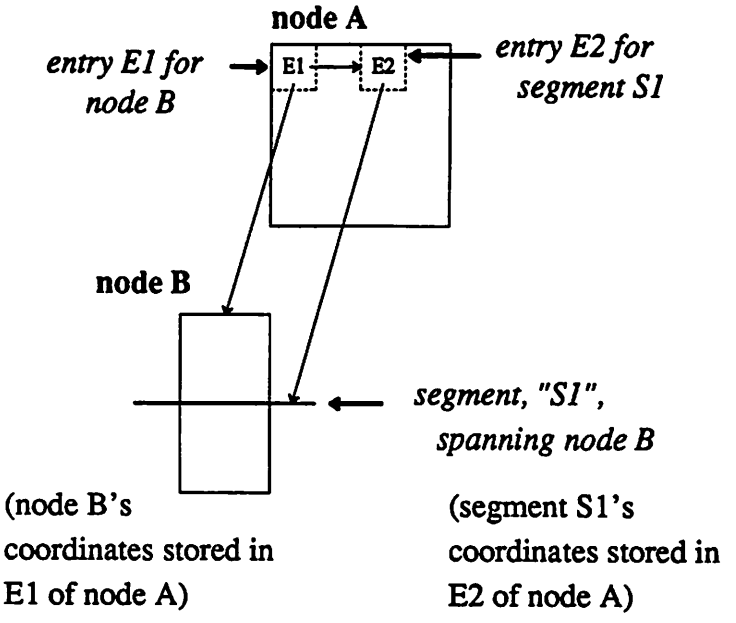


Figure 2: A spanning rectangle in an SR-Tree

require expansion. Moreover, each non-leaf node must enclose all its descendants, and therefore each node along the path from a leaf node to the root node is expanded, if necessary, to minimally enclose the newly inserted rectangle. Such node expansion may also be required for rectangles inserted on non-leaf nodes. Since spanning rectangles are stored in the parent of a spanned node (associated with the branch of the spanned node), if a newly inserted rectangle R is inserted onto a non-leaf node N, although R does not span N in any dimension it may not be properly enclosed by the region associated with N. In that case, N and all its ancestors must be expanded to enclose R.

3.1.2. Node Splitting Algorithm

The SR-Tree leaf node splitting algorithm is the same as that of the original R-Tree node splitting algorithm. For splitting non-leaf nodes, the SR-Tree algorithm divides branches into the two sibling nodes as in the original R-Tree algorithm, and then associates each spanning rectangle with the branch to which it corresponds. Nodes are expanded to minimally enclose the regions covered by the branches within each node.

The algorithms stated above for insertion and node splitting are not complete, as they require further mechanisms to deal with the possible *promotion* (moving to a higher level node) or *demotion* (moving to a lower level node) of spanning rectangles. These issues arise in the following situations:

(1) When a node *N* splits into two nodes *N1* and *N2*, spanning rectangles on node *N* may require a promotion to their parent node, since after the split some spanning rectangles may span either *N1* or *N2*.

(2) A node whose region has expanded due to the insertion of a new rectangle may break former spanning relationships and thus require spanning rectangle demotions.

Solutions to handle each of the two cases mentioned above as they occur are as follows:

(1) When a node *N* splits into two new nodes *N1* and *N2*, check if any spanning rectangle on *N* spans either *N1* or *N2*. For each one that does, remove it from its present node and insert it onto its parent node, and link it to the list associated with the branch of the node which it spans.

(2) On each node that has been expanded, determine whether there exist any displaced spanning rectangles (i.e., formerly spanning rectangles which no longer span any branch on the node). For each such rectangle, remove it from the node and reinsert it into the index.

In order to better handle randomly ordered input, we define a *hybrid* SR-Tree insertion algorithm as one which inserts rectangles whose maximum length in any dimension is less than or equal to a parameter *Z* (such rectangles are hereafter referred to as *"small"*) according to the standard R-Tree algorithm, and applies the SR-Tree insertion algorithm otherwise. The motivation for this insertion scheme is that when rectangles are inserted in random order, it is desirable to insert the "small" rectangles in the leaf nodes as much as possible. Otherwise, if the SR-Tree insertion algorithm were applied to all rectangles then it is possible that some "small" rectangles may be placed in higher-level (non-leaf) nodes and also that some "large" rectangles may be stored in leaf nodes. Although such non-optimal placement of rectangles would be partially remedied by subsequent rectangle promotions and demotions as the index grows, *promotable* rectangles in leaf nodes that have not been recently split may not get promoted. By always placing "small" rectangles in the leaf nodes, it is more likely that the index will evolve in a nearly optimal configuration, even in the case of randomly ordered input.

As a further optimization, after all insertions are performed, the entire index may be postprocessed by first removing all of the rectangles whose maximum length in any dimension is greater than Z from the leaf nodes, and then reinserting those rectangles into the index. This postprocessing step ensures that all rectangles are stored at the proper level in the index regardless of their original insertion order. However, since it is a form of global reorganization, we did not perform this postprocessing algorithm in our performance experiments.

3.1.3. Search Algorithm

The SR-Tree search algorithm is similar to that of the original R-Tree. It descends the index depth-first, descending only those branches that intersect the given search rectangle S until the qualifying data records are found in a set of leaf nodes. In addition, at each node encountered during the search of the index, *all* spanning rectangles are examined to determine if they have a non-zero intersection with S. Since spanning rectangles contained by a node N are wholly contained by N, all spanning rectangles in the index that have a non-zero intersection with S are guaranteed to be found by the search algorithm.

3.2. SB-Tree

The S-Tree adaptation of the B-Tree that we shall present in this section is based on the B+-Tree [COME79] which stores both point and interval data in one dimension, and is hereafter referred to as the SB+-Tree. The idea of the SB+-Tree is that it stores interval data in both the leaf and non-leaf nodes, and point data in the leaf nodes. Alternatively, one may base a S-Tree on the original B-Tree proposal [BAYE72], hereafter referred to as the SB-Tree, in which point data records are stored in both the leaf and non-leaf nodes, and interval data is stored in the non-leaf nodes. In the original B-Tree proposal, point data records in the non-leaf nodes act as separators between branches to other lower-level nodes, whereas the B+-Tree stores separator values and branch pointers in the non-leaf nodes and data records in the leaf nodes. In both the SB+-Tree and SB-Tree, intervals are stored at the highest level in the index such that the interval spans at least one sub-tree rooted at the node.

Comparing our insertion algorithm for the SR-Tree to that of the SB+-Tree, we have illustrated a minor difference in the manner that spanning objects may be inserted on non-leaf nodes. In the SR-Tree, a spanning object is attached to a linked list that is associated with a single spanned branch, whereas in a

SB+-Tree a spanning object is linked to all of its spanned branches. This distinction was made to illustrate the time/space tradeoff between these two approaches. In particular, we reduce space overhead in SR-Trees caused by spanning objects at a slight increase in computational overhead. If an interval spans more than one branch in an SR-Tree, its search algorithm checks *all* spanning objects present on a node for intersection with the specified search argument *A*. On the other hand, in a SB+-Tree, the search algorithm checks only those spanning objects for intersection with *A* which are linked to branches that have non-zero intersection with *A*. Since our metric for search performance was the number of page accesses per search in our performance experiments (discussed in Section 4), the difference in computational cost between these two approaches is not significant. In any S-Tree index, whether a spanning object is linked to one or more than one spanned branches on a node is strictly a time/space tradeoff, as either approach is viable.

3.2.1. Insertion Algorithm

When inserting point data, the SB+-Tree algorithm is the same as that of the original B+-Tree. When inserting an interval *I*, the SB+-Tree algorithm proceeds by first descending the index depth-first, starting from the root node. If *I* spans one or more branches on the current node, *I* is installed on the node and is linked to each branch that it spans. The insertion algorithm does not descend any of the spanned branches. Each remaining branch (not spanned by *I*) is descended, and at each node so encountered, the above algorithm is applied.

3.2.2. Node Splitting Algorithm

When a SB+-Tree node *N* is split, if *N* does not contain any interval data, the algorithm is the same as that of the original B+-Tree. Otherwise, the algorithm divides *N*'s branches between *N* and its new sibling node as in the original algorithm, and then distributes each interval onto the appropriate node(s) according to which node contains the branch(es) that the interval spans. After this step, if there are any intervals that span either of the split siblings, those intervals are promoted to the parent node and are associated with their corresponding spanned branch.

3.2.3. Search Algorithm

Given a search argument consisting of a point value, the search algorithm proceeds exactly as in the corresponding B+-Tree search algorithm. If the search argument is an interval I, the search algorithm proceeds by descending the index depth-first. At each non-leaf node, the intervals that intersect I are returned as qualifying records. At each leaf node, both the points and intervals that intersect I are returned.

4. Performance Experiments

We implemented SR-Trees that use the *hybrid* insertion scheme presented earlier, and that do not use the postprocessing optimization. We performed a series of experiments to compare the performance of the SR-Tree against that of the R-Tree. The input to our experiments were sets of data objects consisting of intervals in two dimensions, i.e., rectangles. For each trial, the input consisted of a high proportion of relatively "small" rectangles, and a low proportion of "big" rectangles. The small rectangles had widths and heights that were uniformly distributed in the range of [100, 1000], while the big rectangles had dimension lengths uniformly distributed over [40000, 60000]. The entire data domain U consisted of a space bounded by [0, 100000] in two dimensions. Center-points for the rectangle data were uniformly distributed over U.

There were five sets of experiments. The first four experiments compared the search performance of an SR-Tree to that of an R-Tree for each of the following types of input data:

(1) randomly ordered input (unsorted both in terms of size and spatial location),

(2) input spatially sorted by the upper horizontal ordinate, i.e., by increasing X2 values given that we specified rectangles by two points: (X1, Y1, X2, Y2),

(3) input partitioned by size (the small rectangles were inserted before the big rectangles in two *batches*), where within each batch the input was inserted in a random order, and

(4) input partitioned by size (the small rectangles were inserted before the big rectangles), where within each batch the input was spatially sorted as in (2).

In the first four experiments, both the SR-Tree and R-Tree used a level zero (leaf) node size of 1 Kb, and both doubled the node size at each successive level of the index, i.e., level one nodes had a size of 2 Kb, level two nodes were 4 Kb, etc. The fifth experiment compared an SR-Tree that doubled its node size at each successive level (as in the first four experiments) to an SR-Tree that used a fixed node size of 1 Kb

throughout, when processing spatially sorted, single batch input.

The hybrid SR-Tree insertion algorithm was implemented because it is a simple optimization to employ for input that is unsorted by size, as in experiments one and two. For the parameter Z we used the value of 1000, so that all of the small rectangles would be inserted in the leaf nodes. The hybrid insertion scheme had virtually no effect on SR-Tree performance in the two-batch input cases (experiments three and four), and provided a moderate improvement for SR-Tree performance in experiments one and two.

Within each of the 5 experiments, 16 sets of input were processed. In each set we varied the number of records indexed and the ratio of big to small rectangles. The number of small and big rectangles in each of the 16 sets of input are described in Table 1. Hereafter, when we refer to a database by a single capital letter, we refer collectively to the four databases that share the same number of small rectangles. For example, "database A" refers collectively to databases A1, A2, A3, and A4.

The first phase of each experiment consisted of inserting the data into the respective indexes being compared. After the insertion phase, 100 random searches were performed on each index, where each

Database Name	Big/Small Ratio	# Small Rectangles	# Big Rectangles
A1	0.2 %	45 K	100
A2	2.2 %	45 K	1000
A3	11.1 %	45 K	5000
A4	22.2 %	45 K	10000
B1	0.2 %	60 K	133
B2	2.2 %	60 K	1333
B3	11.1 %	60 K	6667
B4	22.2 %	60 K	13333
C1	0.2 %	75 K	167
C2	2.2 %	75 K	1667
C3	11.1 %	75 K	8333
C4	22.2 %	75 K	16667
D1	0.2 %	90 K	200
D2	2.2 %	90 K	2000
D3	11.1 %	90 K	10000
D4	22.2 %	90 K	20000

Table 1: Characteristics of databases
used as input to experiments

search argument consisted of a square with edge length of 1000 whose center-point was uniformly distributed over [0, 100000]. The same sequence of random searches was repeated for all experiments. The measurements taken included the average number of index page accesses per search. From those results, we calculated the percentage of improvement in search performance of the SR-Tree with respect to the R-Tree for the first four experiments. For the fifth experiment, we measured the same statistic to compare the search performance improvement provided by the doubling node size SR-Tree with respect to the fixed node size SR-Tree.

In all five experiments, to normalize the results for comparative purposes, both indexes used the same page layout. Therefore, in all of the trials, the utilized sizes of the SR-Tree indexes were within 1% of their corresponding R-Tree counterparts, since both used the same page layout and indexed identical data.

4.1. Results of Performance Experiments

The first experiment processed randomly ordered (unsorted) input, and the results of that experiment are shown in Graphs 1 and 2. In Graph 1, the percentage of improvement in search performance provided by the SR-Tree index with respect to the R-Tree is plotted as a function of the ratio of big/small rectangles, for each of the four databases (A, B, C, and D). In Graph 2, the same statistic is plotted as a function of the database size, for each of the four big/small rectangle ratios. Graph 1 shows that the greatest performance improvement occurred at the big/small ratio of 0.2%, and then decreased as the ratio increased. The performance improvement was in the range of 0.2%-40%. In Graph 2, it is clear that the databases with big/small ratios of 0.2% and 2.2% experienced much larger performance improvements that those with ratios of 11.1% and 22.2%. The results in both of these graphs show that the performance improvement increases with the size of the database. Since the input consisted of mostly small rectangles that were inserted in random order over a large space ([0, 100000]), the sizes of nodes tended to be rather large. This resulted in most of the big rectangles being stored in the leaf nodes, and only a few were stored in the non-leaf nodes since a small number of them spanned any node(s). The more big objects stored in the non-leaf nodes, the greater the potential for performance improvement with respect to an R-Tree which stores all objects, both big and small, in its leaf nodes. Since a few big objects were stored in the non-leaf nodes and the rest of the big objects were stored in the leaf nodes, there was a modest performance improvement

when the big/small ratio was very low, and this improvement diminished as the big/small ratio increased.

The results of the second experiment involving sorted input are presented in Graphs 3 and 4, which plot the relative search performance improvement in an analogous fashion to Graphs 1 and 2. These graphs show that the relative improvement ranged between 1%-80%. As in the first experiment, the performance improvement was inversely related to the big/small ratio, and directly related to the database size. The significant difference between the results of the first two experiments is that the performance improvement is approximately doubled in the case of the spatially sorted input. This is because the spatially sorted input caused the average node size to be relatively small, thereby allowing more big objects to be stored in the non-leaf nodes. In the first experiment, since the input was inserted in a random order the non-leaf nodes tended to be quite large in area. The large area of these nodes prevented spanning objects from occupying higher levels of the index, which would enhance the relative benefit of SR-Trees over R-Trees.

In the third and fourth experiments, the input was inserted in two *batches*, i.e., first the small rectangles and then the big rectangles. In the third experiment, each batch was unsorted. The relative performance improvement in this experiment ranged between 2%-13%. The fourth experiment was similar to the third, except that each batch was sorted. The relative improvement in this experiment ranged between 4%-72%. Graphs of the results of experiments three and four are not shown for the sake of brevity, as their results were similar in form to Graphs 1 through 4, and differed only in magnitude.

The fifth experiment compared the performance of the variable node size SR-Tree that doubles its node size at each successive level of the index (leaf nodes are 1 Kb) to a variation of the SR-Tree that used a fixed node size of 1 Kb at all levels of the index. The input consisted of the spatially sorted input that was also used in the second experiment. The results of this experiment are presented in Graphs 5 and 6, which show that the performance improvement ranged between 8%-45%. These results indicate that doubling the node size at successively higher levels in the index (as is done in the "standard" SR-Tree) provided a considerable performance improvement over the fixed node size SR-Tree. This effect is due to the greater node fanout that was present in the higher-level nodes of the index.

Reviewing the results of the first four experiments, it is clear that the SR-Trees had a greater performance advantage over R-Trees in the spatially sorted input cases as opposed to the spatially unsorted input cases. Spatially sorted input is advantageous for SR-Trees because the big rectangles are more likely to be

inserted in higher-level nodes, since the average non-leaf node size would be smaller than that in an SR-Tree that processed the same data collection inserted in a random order.

Our observations regarding the effects of spatially sorted versus randomly ordered input led us to conclude that SR-Trees would benefit from a "packing" algorithm similar to that proposed by Roussopoulos and Leifker [ROUS85]. The algorithm of [ROUS85] builds an R-Tree by successively applying a nearest neighbor relation to group objects in a node after the set of objects has been sorted according to a spatial criterion. This algorithm is applied level-by-level, starting from the leaf level, and fills each node to capacity. The drawback to this scheme for our purposes is that it is a static method, whereas the SR-Tree is designed to be a dynamic index. Although spatially sorting the input was a crude approximation to the packing algorithm of [ROUS85], it provided a dramatic increase in the performance of SR-Trees.

5. Lop-Sided Trees

In this section, we illustrate the utility of our third tactic: the Lop-Sided index. Suppose there is a standard index (i.e., balanced, and all data are stored in the leaf nodes) that consists of one root node at level one and three leaf nodes at level zero, and a Lop-Sided Tree index (unbalanced, and stores data in nodes at all levels) consisting of one node at level two, one, and zero, respectively, as illustrated in Figure 3. Here, two page reads of the standard index are required to access any one of the leaf nodes. For the Lop-Sided index, three page reads are required to access the leftmost node, two reads for the middle node, and one for the rightmost node. If the query distributions are non-uniform, as for example those shown in Table 2, a substantial reduction in the number of page accesses may be achieved with the Lop-Sided index.

6. Summary and Conclusions

A novel way of storing multi-dimensional interval data by modifying a class of database indexing structures that are based on paged, balanced, multi-way trees has been described. Performance results comparing SR-Trees to R-Trees were presented which demonstrate that S-Trees provide a substantial performance improvement over conventional indexing techniques when the length of the interval data is highly non-uniform. SR-Trees were shown to provide an improvement in search performance over R-Trees when the ratio of big to small objects is small, and the performance improvement was enhanced by spatially presorting the input data.

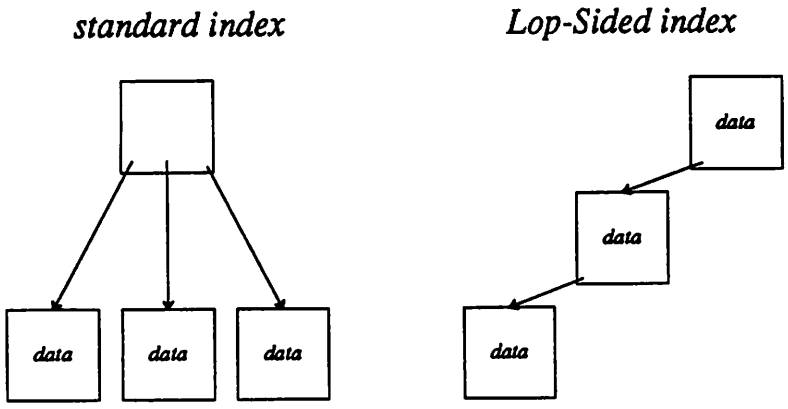


Figure 3: Standard index vs. Lop-Sided index

% accesses to leftmost node	% accesses to middle node	% accesses to rightmost node	avg. # of reads/query (standard)	avg. # of reads/query (Lop-Sided)	performance improvement (%)
0	0.125	0.875	2.0	1.125	43.7 %
0	0.333	0.667	2.0	1.333	33.3 %
0.125	0.25	0.625	2.0	1.5	25.0 %
0.25	0.25	0.5	2.0	1.75	12.5 %

Table 2: Performance improvement provided by Lop-Sided index for various non-uniform query distributions

The plans for our future research are to design and implement other S-Tree index structures and measure their performance. We plan to implement SB+-Trees as well as SR-Trees that index historical data. We also plan to implement a Lop-Sided version of the SR-Tree which supports non-uniform query distributions, and to design and experiment with various adaptive algorithms that tune the degree of "lop-sidedness" according to statistics gathered from the distribution of past queries.

References

[BAYE72] Bayer, R., and McCreight, E., "Organization and Maintenance of Large Ordered Indexes", Acta Informatica, 1, No. 3 (1972), pp. 173-189, Springer-Verlag.

[BENT77] Bentley, J.L., "Algorithms for Klee's Rectangle Problems", Computer Science Department, Carnegie-Mellon University, Pittsburgh.

[COME79] Comer, D., "The Ubiquitous B-Tree", ACM Comp. Surv. 11, No. 2 (June 1979).

[EAST86] Easton, M., "Key-Sequences Data Sets on Indelible Storage", IBM Journal of Research and Development, 30, 3, (May 1986).

[EDEL80] Edelsbrunner, H., "Dynamic Rectangle Intersection Searching", Institute for Information Processing Rept. 47, Technical University of Graz, Graz, Austria.

[GUTT84] Guttman, A., "R-Trees: A Dynamic Index Structure for Spatial Searching", Proceedings of ACM SIGMOD International Conference on Management of Data, Ed. B. Yormark. Association for Computing Machinery. Boston, MA: June 1984.

[LOME89] Lomet, D. and Salzberg, B., "Access Methods for Multiversion Data", Proc. 1989 ACM-SIGMOD Conference on Management of Data, Portland, Ore., June 1989.

[MCCR85] McCreight, E., "Priority Search Trees", SIAM J. Comput. 14, 2 (May).

[PREP85] Preparata, F., and Shamos, M., "Computational Geometry, An Introduction", Springer-Verlag Publishing Co., 1985.

[ROUS85] Roussopoulos, N., and Leifker, D., "Direct Spatial Search on Pictorial Databases using Packed R-Trees", Proc. 1985 ACM-SIGMOD Conference on Management of Data, Austin, TX, May 1985.

[SAME89] Samet, H., "The Design and Analysis of Spatial Data Structures", Addison-Wesley Publishing Co., 1989.

[SARN86] Sarnak, N. and Tarjan, R., "Planar Point Location Using Persistent Search Trees", Comm. ACM, 29, 7 (July).

[SELL87] Sellis, T., Roussopoulos, N., and Faloutsos, C., "The R+-Tree: A Dynamic Index for Multi-Dimensional Objects", Proc. 1987 VLDB Conference, Brighton, England, Sept. 1987.

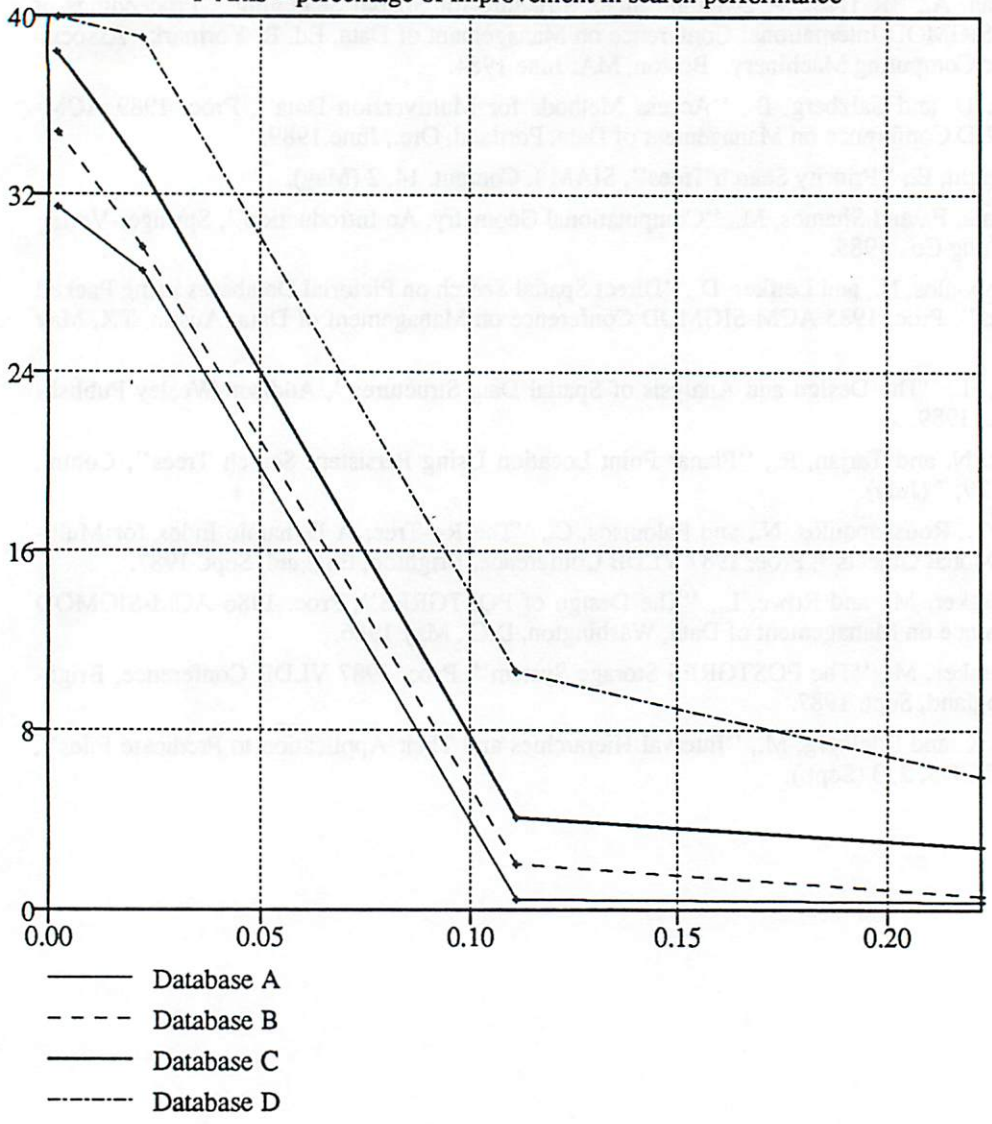
[STON86] Stonebraker, M., and Rowe, L., "The Design of POSTGRES", Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, D.C., May 1986.

[STON87] Stonebraker, M., "The POSTGRES Storage System", Proc. 1987 VLDB Conference, Brighton, England, Sept. 1987.

[WONG77] Wong, K. and Edelberg, M., "Interval Hierarchies and Their Application to Predicate Files", ACM TODS, 2, 3 (Sept.).

PERCENTAGE IMPROVEMENT IN SEARCH PERFORMANCE
OF SEGMENT R-TREES OVER R-TREES,
AS A FUNCTION OF THE RATIO OF BIG/SMALL RECTANGLES

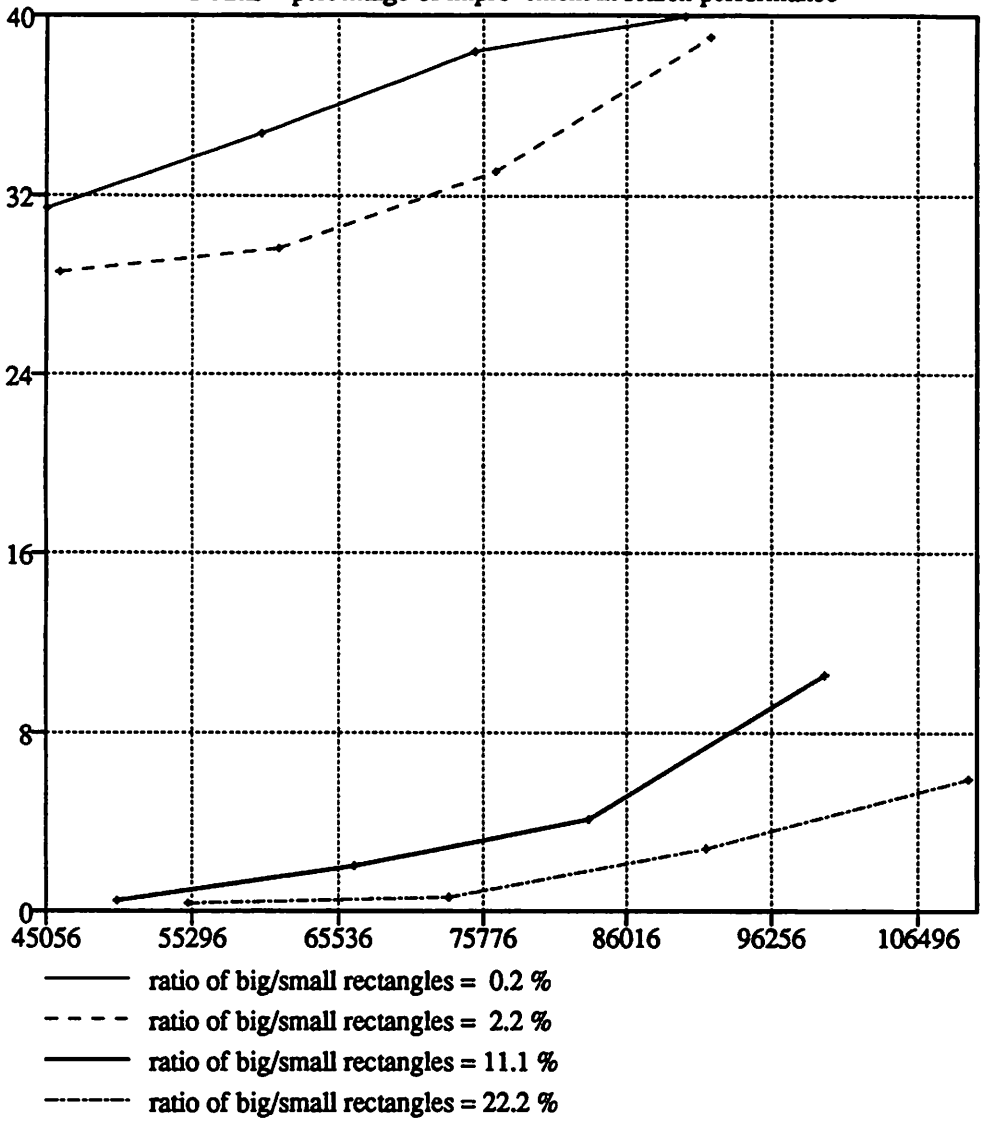
X Axis = ratio of "big" to "small" rectangles
Y Axis = percentage of improvement in search performance



Graph 1: single batch input, randomly ordered

PERCENTAGE IMPROVEMENT IN SEARCH PERFORMANCE
OF SEGMENT R-TREES OVER R-TREES,
AS A FUNCTION OF THE DATABASE SIZE

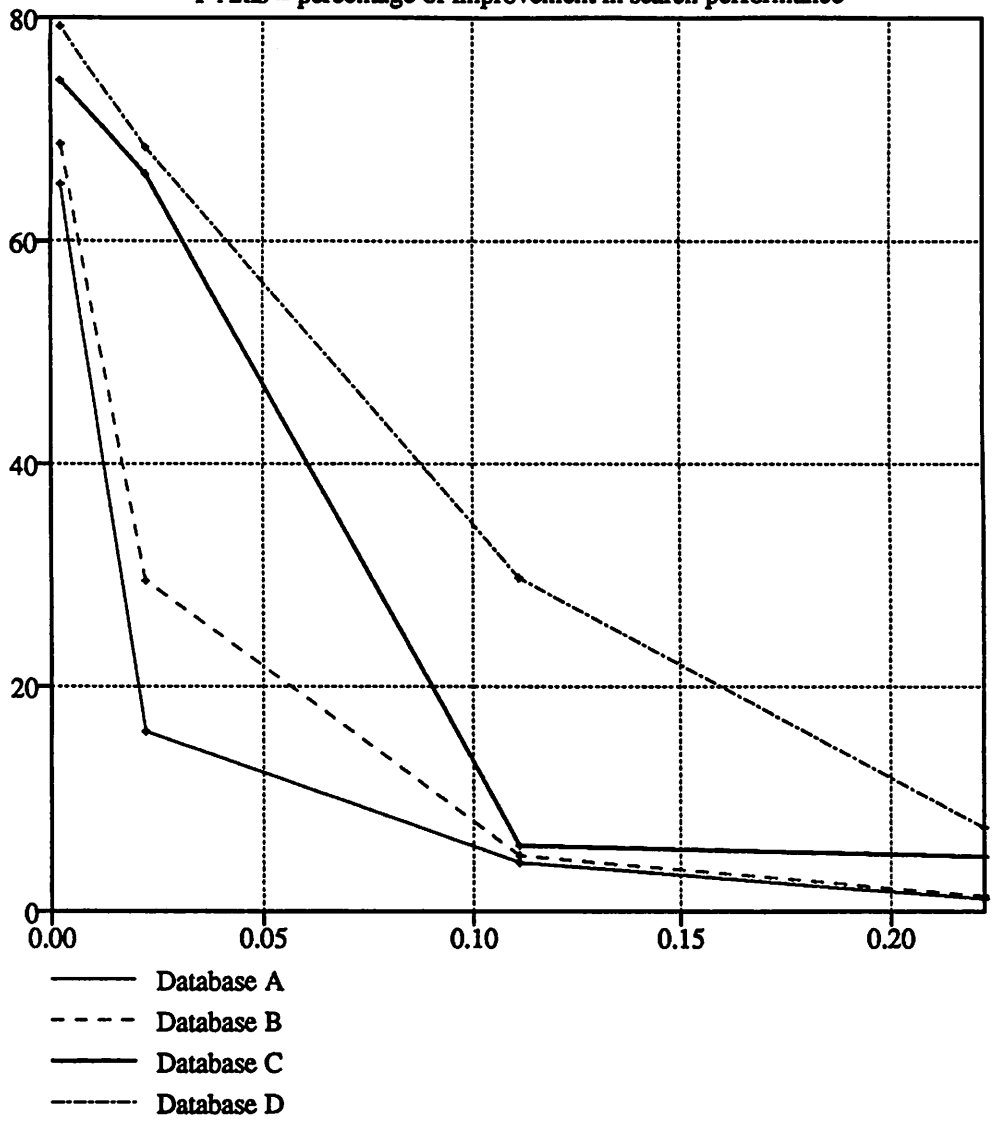
X Axis = database size (no. of records)
Y Axis = percentage of improvement in search performance



Graph 2: single batch input, randomly ordered

PERCENTAGE IMPROVEMENT IN SEARCH PERFORMANCE
OF SEGMENT R-TREES OVER R-TREES,
AS A FUNCTION OF THE RATIO OF BIG/SMALL RECTANGLES

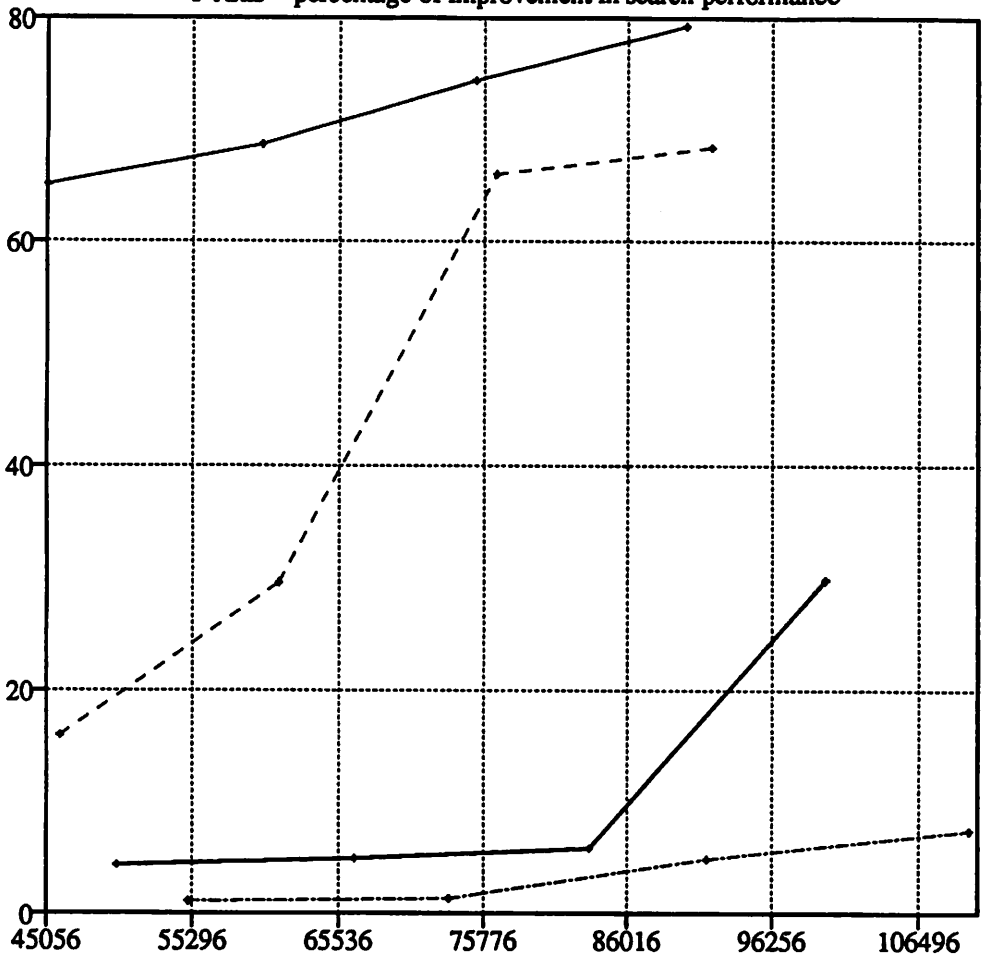
X Axis = ratio of "big" to "small" rectangles
Y Axis = percentage of improvement in search performance



Graph 3: single batch input, sorted by X2

PERCENTAGE IMPROVEMENT IN SEARCH PERFORMANCE
OF SEGMENT R-TREES OVER R-TREES,
AS A FUNCTION OF THE DATABASE SIZE

X Axis = database size (no. of records)
Y Axis = percentage of improvement in search performance

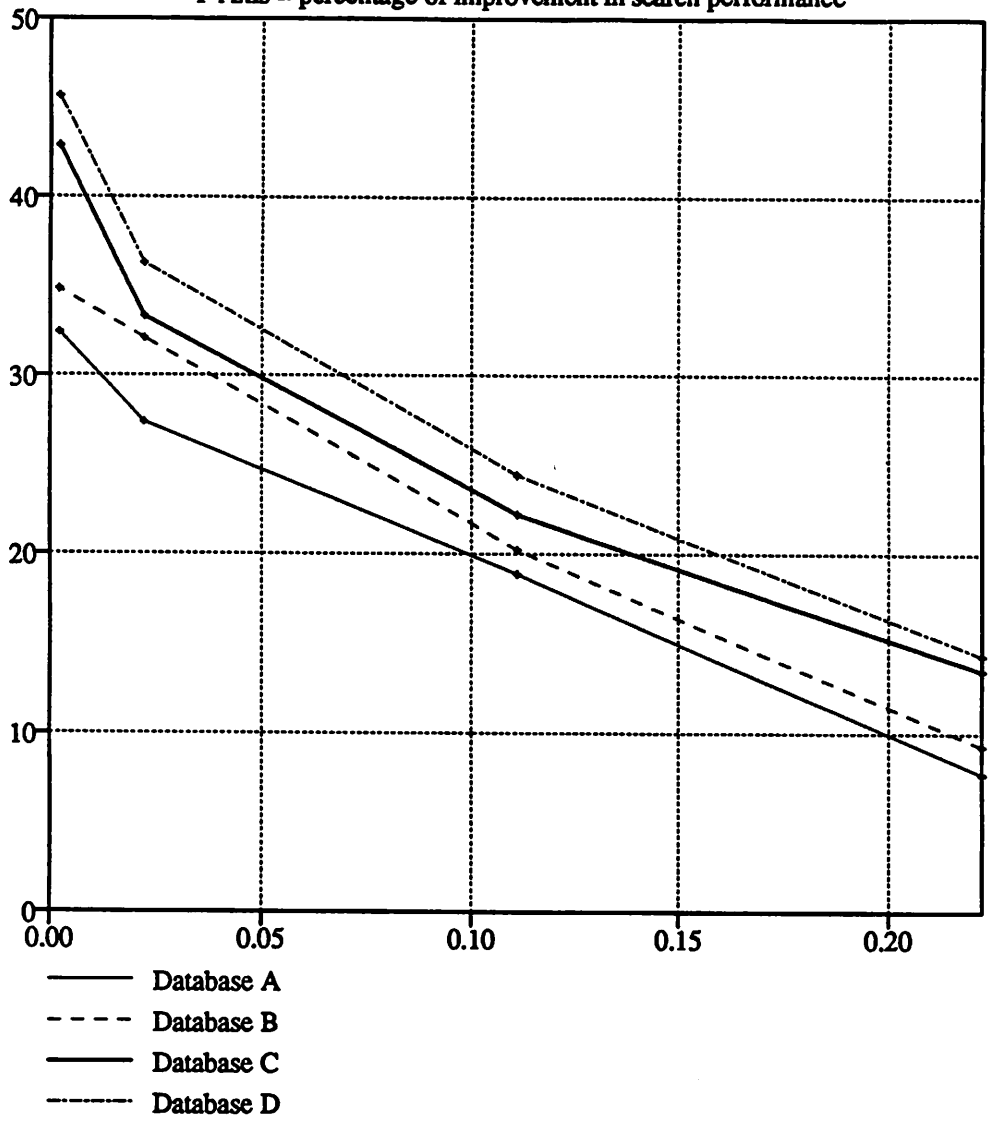


——— ratio of big/small rectangles = 0.2 %
- - - - ratio of big/small rectangles = 2.2 %
——— ratio of big/small rectangles = 11.1 %
—·—·— ratio of big/small rectangles = 22.2 %

Graph 4: single batch input, sorted by X2

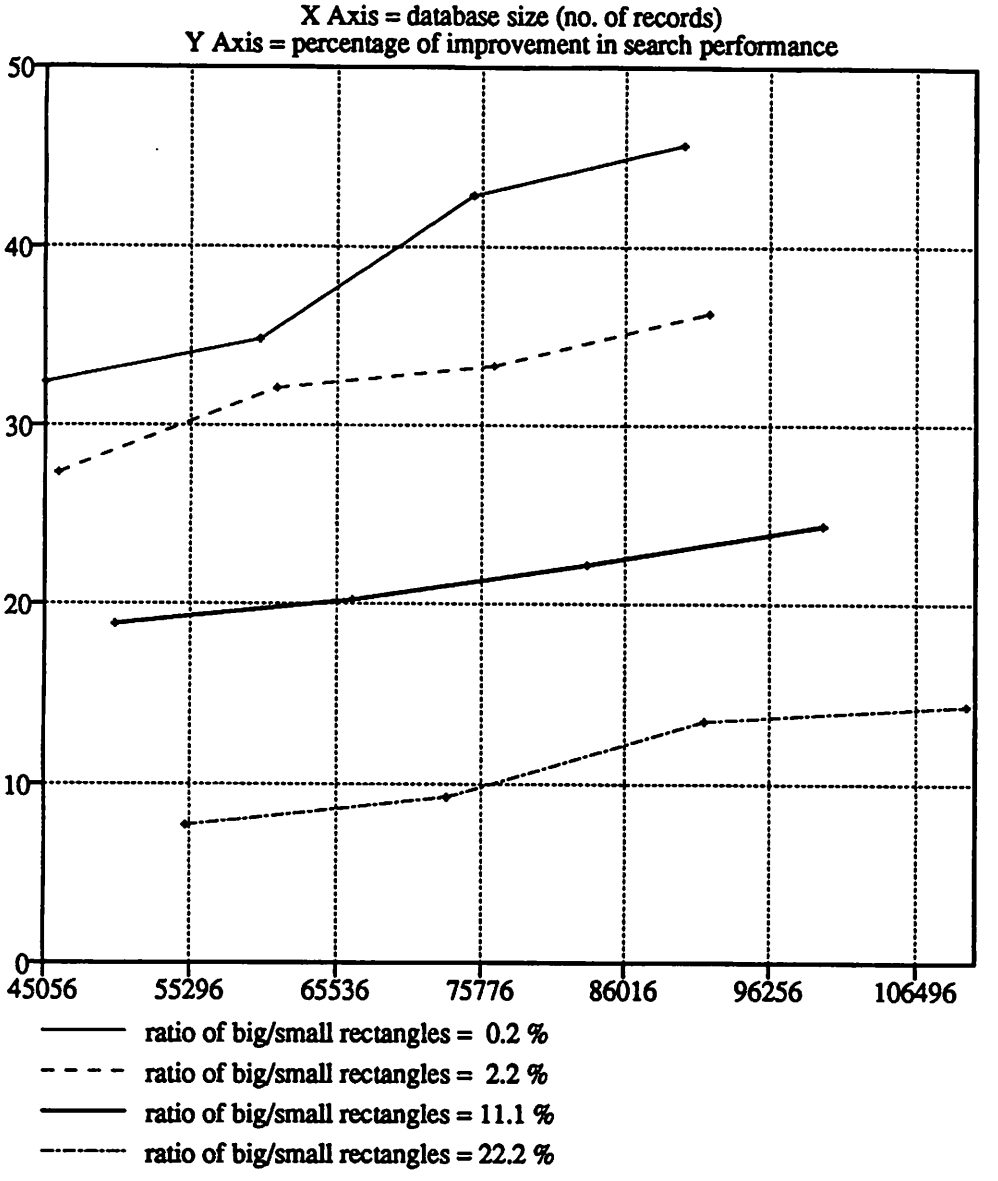
PERCENTAGE IMPROVEMENT IN SEARCH PERFORMANCE
OF DOUBLING VS. FIXED PAGE-SIZE SEGMENT R-TREES,
AS A FUNCTION OF THE RATIO OF BIG/SMALL RECTANGLES

X Axis = ratio of "big" to "small" rectangles
Y Axis = percentage of improvement in search performance



Graph 5: single batch input, sorted by X2

PERCENTAGE IMPROVEMENT IN SEARCH PERFORMANCE
OF DOUBLING VS. FIXED PAGE-SIZE SEGMENT R-TREES,
AS A FUNCTION OF THE DATABASE SIZE



Graph 6: single batch input, sorted by X2