Segment Indexes: Dynamic Indexing Techniques for Multi-Dimensional Interval Data

Curtis P. Kolovson †
Michael Stonebraker ‡

† Database Technology Department
Hewlett-Packard Laboratories
1501 Page Mill Road, Building 3U
Palo Alto, CA 94304

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

ABSTRACT — We propose new indexing techniques for interval data in $K \geq 1$ dimensions consisting of a set of extensions to a class of database indexing structures. These techniques are useful for improving index search performance for spatial data composed of multi-dimensional intervals that have non-uniform length distributions. Interval data collections having non-uniform length distributions are likely to occur in practice, and may be typical of historical data collections in which tuples represent intervals in the time dimension. We present these indexing techniques, illustrate how they may be applied to the R-Tree index, and provide the results of performance experiments.

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, $K \geq 1$. Currently, several proposals have been made for data structures that support efficient access to large collections of multi-dimensional interval data. We categorize these proposals into two significant classes:

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

(2) Disk oriented 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 structure is contained in main memory, and none have been extended to multi-way (n-ary) trees that may be efficiently paged onto secondary storage.

Several indexing techniques have been proposed for database management systems to deal with 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 original R-Tree [GUTT84] and its variants such as the R+-Tree [SELL87] and R*-Tree [BECK90] store all interval data records in the leaf nodes. None of these indexing techniques are particularly well-suited for interval data whose length distribution is highly non-uniform, e.g., a large proportion of "short" intervals and a small 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 with 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). One of the goals of this work is to find efficient indexing techniques to support queries on historical data [STON86, STON87].

In this paper, we combine aspects of the memory resident data structures from Computational Geometry and disk oriented indexing structures from Database Management Systems 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 a given time. As an example, we describe how aspects of the Segment Tree [BENT77] may be merged with features of the R-Tree [GUTT84], and refer

This research was performed at the University of California, Berkeley, and was partially supported by the Army Research Organization Grant DAAL03-87-0083, by the Defense Advanced Research Projects Agency through NASA Grant NAG2-530, and by a Hewlett-Packard Laboratories Resident Fellowship.

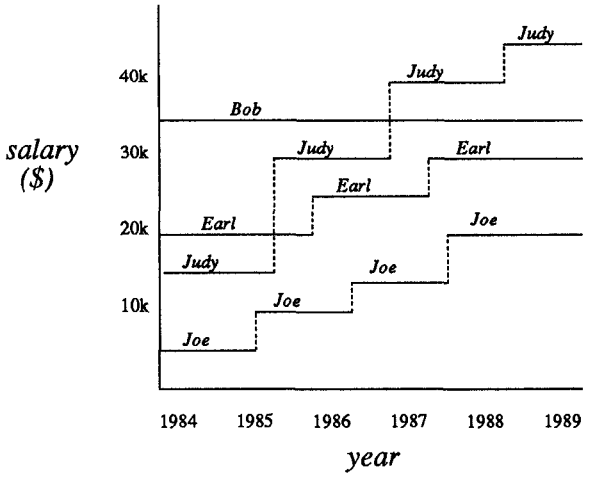


Figure 1: Historical data: Employee salaries as a function of time

to the new index as a *Segment R-Tree*, or more succinctly as the *SR-Tree*. The R-Tree is a K-dimensional variation of the B-Tree [BAYE72] which indexes data consisting of intervals in K dimensions, $K \geq 1$. Both the R-Tree and SR-Tree may be used for indexing line *segment* data (intervals in one dimension) or spatial data (intervals in K dimensions, $K > 1$).

The remainder of this paper proceeds as follows. Section 2 presents our tactics and the motivation for our research. Section 3 describes the algorithms used by the SR-Tree. Section 4 presents the notion of adaptable pre-constructed indexes, which we refer to as *Skeleton Indexes*. Section 5 discusses the results of performance experiments which compare the performance of the SR-Tree and Skeleton SR-Tree presented in Sections 3 and 4, respectively, to that of the R-Tree. Section 6 presents our summary and conclusions.

2. The Segment Index Approach

The *Segment Index* approach combines aspects of the memory resident Segment Tree data structure [BENT77] with those of a class of database access methods that are based on paged, multi-way, tree-structured indexes. The Segment Tree data structure stores line segments in a binary tree by storing the segment end-points in the leaf nodes, and then associates each interval with the highest level node N that *spans* the values corresponding to the left and right children of N. We say that an interval I_1 *spans* another interval I_2 if $I_1.low\_limit \leq I_2.low\_limit$ and $I_1.high\_limit \geq I_2.high\_limit$.

In this section, we present the tactics used to convert a paged, multi-way, tree-structured index into a Segment Index, and some motivations for this new indexing approach.

2.1. Tactics

This research investigates three major modifications to tree-structured indexes to support efficient search operations on multi-dimensional interval data:

(1) Index records may be stored in non-leaf nodes.

(2) The index node size may vary.

(3) The index may be pre-constructed based on an estimate of the input data distribution, and later made to adapt to the actual input data distribution.

2.1.1. Storing Index Records in Non-Leaf Nodes

The first tactic is to allow index records to be stored in both non-leaf and leaf nodes, as opposed to only in the leaf nodes as is conventionally done. Intervals are placed in non-leaf nodes according to a certain criterion; namely, an interval I is stored in the highest level node N of a tree-structured index such that I spans at least one of the intervals represented by N's child nodes. By organizing intervals in this way, an index is well-suited to processing queries that request all intervals that contain a given point, or that intersect a given interval range.

In addition, there are advantages to this approach that are specific to certain spatial indexing techniques, such as the R-Tree and R+-Tree. In the case of R-Trees which allow overlap between node regions and do not partition input data, by storing "long" intervals in higher level nodes there would be less node overlap because the leaf nodes would contain mostly "short" intervals. Overlapping nodes degrade search performance in R-Trees, since all non-leaf nodes that intersect a given search region must be searched. The problem with storing "long" intervals in leaf nodes is that doing so tends to exacerbate the overlap problem by elongating the nodes that contain them. In the case of R+-Trees which partition data in order to avoid node overlap, by storing "long" intervals in higher-level nodes the lower-level nodes would have fewer replicated index records (fewer partitioned intervals). Storing a "long" interval in a higher level node as a single index record is more space efficient than the R+-Tree approach of breaking it up into many sub-intervals and storing them in a set of leaf nodes.

2.1.2. Varying the Index Node Size

To support the first tactic, it may be desirable to have larger node 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 node in a Segment Index, a non-leaf node with a large number of *external* index records will have reduced fanout. In order to maintain high fanout in such an index, it is desirable to increase the size of a node at each successively higher level of the index.

2.1.3. Skeleton Indexes

A *Skeleton Index* is an adaptable pre-allocated Segment Index which is built using estimates of the input size (number of tuples) and value distribution. After the initial index is constructed, the

index dynamically adapts to the input data. The motivation for Skeleton Indexes is to provide a more regular decomposition of the regions covered by the non-leaf nodes of the index, so that "long" intervals will be more likely to span lower level nodes, as discussed in Section 2.1.1.

2.2. Motivation for Segment Indexes

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

(1) improve the performance of spatial indexing structures,

(2) provide an efficient indexing technique for historical data,

(3) efficiently index one-dimensional intervals and point data.

We have already discussed the first two motivating influences. The third motivation is to efficiently index both *interval* and *point* data in a single index. One instance where such a need arises is when historical data consists of both *time range* and *event* data. A time range data item consists of an interval in the time dimension and a point in K other dimensions, $K \geq 1$, as described above. An event data item is a point in all dimensions.

Another case where variable length intervals and point data may be intermixed is if database indexes are used to support *rules* by means of storing rule locks in an index. A rule activation may be triggered by an input data value falling within a specified interval, or if the input data value equals a specified value. For example, suppose there is a set of rules affecting office assignments, including the following two rules:

Rule 1: if (EMP.salary > \$10K and
EMP.salary ≤ \$20K) then
EMP.office_type := "office has at least 1 window"

Rule 2: if (EMP.salary = \$100K) then
EMP.office_type := "office has at least 4 windows"

In this example, Rule 1 is triggered if an employee's salary falls within a specified interval, whereas Rule 2 is triggered if an employee's salary equals a specific amount.

One approach to managing rule locks is to store *index stub records* that correspond to rule predicates in an index [STON90]. Index stub records are marked with rule locks and are installed at both ends of an interval corresponding to a rule predicate, e.g., the *left* and *right* index stub records corresponding to Rule 1 would be stored at values of \$10K and \$20K, respectively. All intervening index records between these stub records must also be marked with the appropriate rule lock. If all index records on a node N are marked with the same rule lock, then the corresponding rule lock is said to *span* node N and therefore may be *promoted* (escalated) to its parent node.

An indexing technique that would be suitable for managing rule locks as described above is the Segment Index adaptation of a one-dimensional R-Tree, which is a special case of the K-dimensional Segment R-Tree presented in the next section.

3. An Example Segment Index

The central concept underlying Segment Indexes is that intervals which span lower level nodes may be stored in the higher level nodes of an index. The manner in which this may be applied to a particular indexing structure depends both on the original structure and the type of data being indexed. In this section, we describe the design of one Segment Index by describing the required modifications to the algorithms for insertion, node-splitting, and search operations. The index is based on the R-Tree, and may be used for indexing historical data (as in Figure 1) or spatial data. In an earlier paper, we reported the results of performance experiments in which the R-Tree was used as an indexing structure for historical data [KOLO89].

3.1. SR-Tree

We define the *SR-Tree* as the Segment Index adaptation of the R-Tree index. In this section we describe the insertion, node splitting, and search algorithms utilized by the SR-Tree. The algorithm descriptions given below are extensions to the original R-Tree algorithms as presented in [GUTT84]. With no loss of generality we present a two-dimensional SR-Tree ($K = 2$), as extensions to the cases of $K = 1$ or $K > 2$ are straightforward.

3.1.1. Insertion Algorithm

To insert an index record R into an SR-Tree, the index is searched top-down, depth-first, beginning with the root node. Each branch index record B (which contains a pointer to a child node) of a node N is tested to determine if the region of the node pointed to by B is spanned by R. If it is, then R is a *spanning index record*, and it is inserted onto node N and linked to the list associated with B, and the insertion of the record is completed. For each branch index record, there is a list of its spanning index records. If R is a 2-dimensional rectangle as opposed to a 1-dimensional line segment, then R qualifies as a spanning index record if it spans B's region in either or both dimensions.

A non-leaf node containing a spanning segment is illustrated in Figure 2. In this figure, node A contains a branch index record labeled $E1$ which contains a pointer to child node B. Since line segment $S1$ spans the region covered by node B, but not that of node A, $S1$ is represented as a spanning index record labeled $E2$ on node A, and $E2$ is linked to the list of spanning index records of branch index record $E1$.

A spanning index record spans the region covered by a *child* of some node N on which it is stored and therefore cannot span the region of N itself. However, a spanning index record may extend beyond a boundary of N (the parent of the spanned node) in one or more dimensions. If that is the case, the data item is *cut* into a *spanning portion* and one or more *remnant portions*, and the remnant portion(s) are inserted into the index.

An example of a segment cut into spanning and remnant portions is illustrated in Figure 3. In this figure, the original segment spans node C, but not C's parent (node A). However, since the segment does extend beyond one border of C's parent node, the segment is cut into a spanning portion (which spans node C and is fully enclosed by C's parent), and a remnant portion (which extends beyond the boundary of C's parent). Since the remnant portion does not span any node, it is stored in leaf node E.

The alternative to cutting index records into spanning and remnant portions is to *stretch* nodes to minimally enclose their

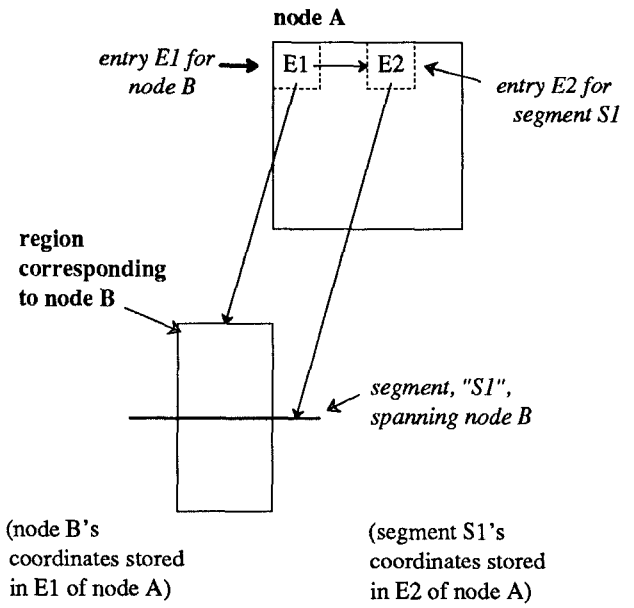


Figure 2: SR-Tree storing a spanning segment

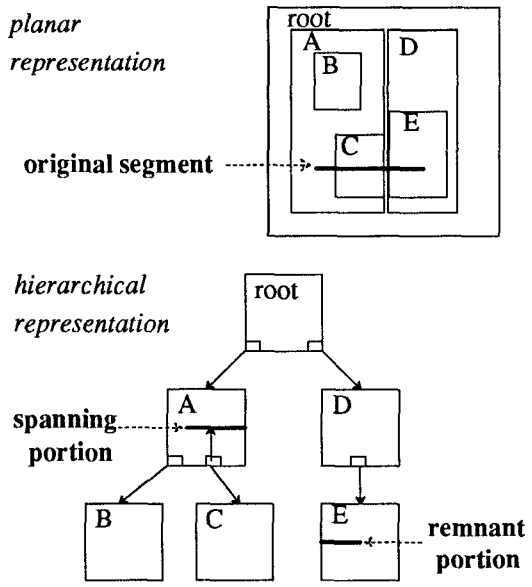


Figure 3: Cutting a segment into *spanning* and *remnant* portions

spanning index records, but this has the disadvantage of degrading the search performance of the index due to increasing node overlap. The disadvantages of cutting index records are: (1) the space overhead of storing potentially more than one index entry per data item, and (2) the need to search the entire index for related spanning/remnant index records when modifying or deleting a single (logical) index record. However, these disadvantages may not be significant since (1) the need for cutting index records arises only in the infrequent event that a spanning index record is not already enclosed by the *parent* of a spanned node, and (2) historical data indexes only need to support insertion and search operations, thus obviating the need for modifying or deleting index records.

If the index record R to be inserted does not span any of the regions of the branches on the root node, the branch B is chosen that requires the least area expansion to fully enclose R.[1] The insertion algorithm is recursively applied to the node pointed to by the selected branch B. If the recursive descent of the index reaches a leaf node L, the index record R to be inserted does not span any non-leaf nodes and will be inserted on node L. After the index record R is inserted on node L, the region covered by each non-leaf node encountered during the recursive descent is expanded (if necessary) to minimally enclose the newly inserted index record.

[1]The strategy of selecting the branch requiring least area expansion is the same as that employed by the original R-Tree, which attempts to minimize the total area covered by the union of the non-leaf node regions.

The algorithm stated above for insertion is not complete, as it requires one further enhancement to deal with the possible *demotion* (moving to a lower level node) of spanning index records. This possibility arises if a node whose region has expanded due to the insertion of a new index record breaks former spanning relationships, thus requiring the demotion of one or more (former) spanning index records. To handle segment demotions, each node that has been expanded is checked to determine whether it has any *demotable* spanning index records (i.e., formerly spanning index records which no longer span any branch on the node). Each such demotable index record is removed from its node and reinserted into the index.

3.1.2. Node Splitting Algorithm

When a node in an SR-Tree has every entry in use and an attempt is made to insert a new entry onto that node, the node is said to *overflow*. When a node overflows, it is split into two nodes and its original contents are distributed between the two new nodes. For leaf nodes, the algorithm for node splitting is identical to that of the original R-Tree algorithm. For non-leaf nodes, there are two differences with respect to the original R-Tree node splitting algorithm. The first difference is that an R-Tree node may overflow due to an attempt to insert a new branch onto an already full node, whereas an SR-Tree node may overflow due to an attempt to insert either a new branch or a spanning index record onto an already full node. The second difference is that if a set of branch entries (pointers to child nodes) are transferred to a new sibling node as a result of a split, the spanning index records that are linked to those branches must also be "carried over" to the new sibling node.

The process of splitting a non-leaf node in an SR-Tree is illustrated in Figure 4. The top of Figure 4 shows a full node (before being split), where some entries are branches (labels begin with **B**) and the other entries are spanning segments (labels begin with S). The bottom of Figure 4 shows the two resulting nodes after the split of the original node. In this figure, the branches are distributed according to the R-Tree node splitting algorithm, and the spanning segments are then transferred to the node that contains the branch that they are linked to.

The algorithm stated above for node splitting is not complete, as it requires one further mechanism to handle the possible *promotion* (moving to a higher level node) of spanning index records. This issue arises when a node N is split and its contents are distributed between N and its new sibling, *N-sibling*. Spanning index records on these two new nodes may need to be promoted to their parent node, since after the split some spanning index records may span N or *N-sibling*. To process index record promotions, after a node N is split, all spanning index records on these nodes are checked to determine if they span the region of N or *N-sibling*. Each one that does is removed from its node, inserted onto its parent node, and linked to the branch of the node which it spans.

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 index records are examined to determine if they have a non-zero intersection with S. Since spanning index records contained by a node N are wholly contained by N, all spanning index records that have a non-zero intersection with S are guaranteed to be found by the search algorithm.

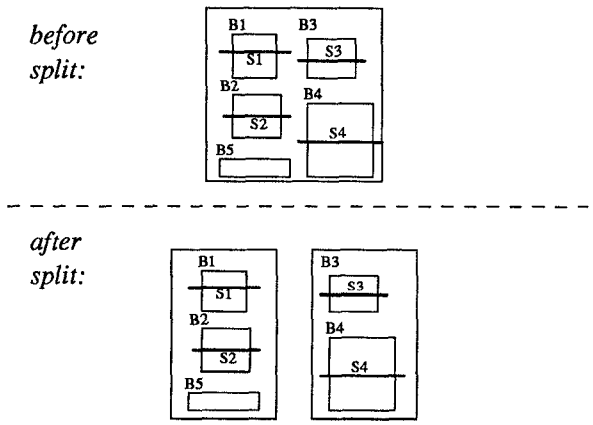


Figure 4: Splitting a non-leaf node in an SR-Tree

4. Skeleton Indexes

The standard R-Tree and SR-Tree insertion algorithms begin with a single node, and perform splits as nodes overflow, similar to a B-Tree. The main strategy used by R-Trees, and hence SR-Trees, with regard to index record placement and node splitting decisions is to minimize the total area covered by the union of the non-leaf node regions. This algorithm is not always optimal for the SR-Tree, however, since the horizontal-to-vertical aspect ratios of the regions covered by the non-leaf nodes are difficult to control. The two difficulties for the SR-Tree that arise as a result of this problem are that (1) nodes may have regions whose aspect ratios are extremely large or small, thus reducing the potential for "long" intervals to span lower level nodes, and (2) nodes may have a high degree of overlap. Both of these problems are sensitive to the order of insertion. In particular, they may be partially alleviated by applying a *packing* algorithm, such as that suggested by [ROUS85]. However, such an approach is a static method which requires that all of the data be available before the index is constructed. Since the SR-Tree is designed to be a dynamic index, an alternative solution to the two aforementioned problems is to use a pre-allocation scheme which we refer to as the *Skeleton SR-Tree*.

A Skeleton SR-Tree is an SR-Tree index which pre-partitions the entire domain into some number of regions. If the input data is uniformly distributed, partitions are of equal size at each level of the index, as illustrated in Figure 5. The number of levels and sub-regions at each level depend on estimates of the number of records to be inserted, the input data distribution, and the node branching factor at each level. At each level, the branching factor of a node depends on the node size and the number of node entries that are reserved for branch entries (as opposed to spanning index records). Once a Skeleton Index is built, it is populated by inserting index records in any order.

A Skeleton SR-Tree is built in a top-down fashion as follows. First, the number of nodes at each level of the index is computed,

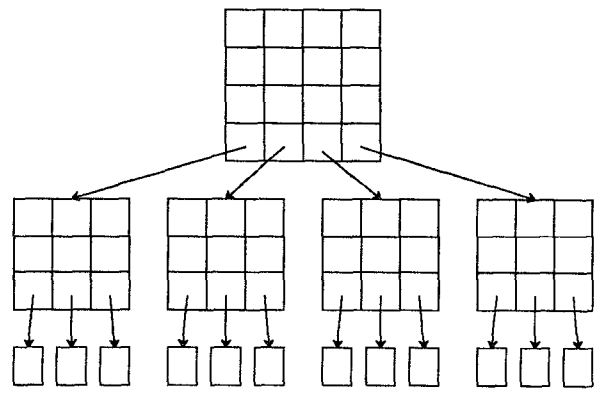


Figure 5: A *Skeleton* SR-Tree

based on the node fanout at each level. The fanout at each level is a function of the node size and the number of node entries that are reserved for node branch entries, as opposed to spanning index records. The number of entries on a node that are reserved for branches may be some fraction of the available entries, e.g. 1/2, 2/3, or 3/4, and is chosen based on the expected number of spanning index records. Assuming the fanout at each level is stored in an array called *fanout* and the expected number of tuples to be inserted is stored in *number_of_tuples*, the number of nodes at each level and number of levels are computed by the following loop (using pseudo-C notation):

```
n = number_of_tuples;
level = 0;              /* level zero is the leaf level */
while (n > 1)
{
  number_of_nodes_level = ⌈ √⌈ n / fanout_level ⌉ ⌉^2 ;
  n = number_of_nodes_level;
  level = level + 1;
}
number_of_levels = level;
```

For simplicity, we round-up the number of nodes at each level to be a number whose square root is integral[2] so that the index may be initially constructed with an equal number of partitions in each dimension. Therefore, the number of partitions in each dimension at level i is the square root of the number of nodes at level i. Once the number of levels and number of nodes at each level of the index are computed, each dimension of the index is pre-partitioned based on the expected distribution of the input in each dimension. If the input data distribution is known in advance, it may be specified by a histogram for each dimension. Given such a set of histograms, the index is constructed one level at a time, in a top-down fashion. At each level of the index, information from the histograms and the number of partitions per dimension are used to determine the partition values in each dimension of the index.

The Skeleton SR-Tree scheme described above works well when the input data distribution is either uniform or has a known distribution. An example showing the partitioning of a Skeleton SR-Tree root node based on a given non-uniform distribution is illustrated in Figure 6.

When the input data distribution is unknown, one approach is to assume uniformly distributed data and build the corresponding *uniform* Skeleton Index, and later adapt it to the actual data through node splitting and merging. An alternative approach to starting with a uniform Skeleton Index is to use a technique which we refer to as *distribution prediction*, which is applicable when it may be assumed that tuples are inserted in random order. The idea of distribution prediction is to buffer the first T tuples in main memory, and compute a histogram of the initial input data in each dimension, and then construct a Skeleton Index based on those histograms. In our experiments, values of T in the range of

[2] We have assumed a two-dimensional Skeleton SR-Tree. If the number of dimensions were D, $D > 2$, then we round-up the number of nodes at each level so that their D-th roots are integral.

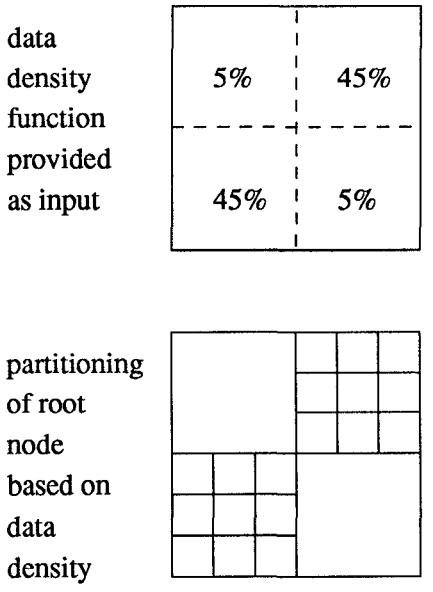


Figure 6: Partitioning a Skeleton SR-Tree root node based on a given non-uniform distribution

5% to 10% of the expected number of tuples to be inserted worked well.

Since distribution prediction may not always predict the exact data distribution, Skeleton Indexes must be *adaptable*. In particular, high-density regions must become *finer grained*, and sparsely populated regions must become *coarser grained*. High-density regions are made finer grained through conventional node splitting, as in the SR-Tree. Sparsely populated regions that are spatially adjacent are merged, or *coalesced*. The frequency of checking for nodes to coalesce may be a parameter (e.g., after every I insertions), and additional measures may be used to restrict the nodes that are potential candidates for coalescing. For example, statistics may be collected to keep track of the L least frequently modified nodes, and only those nodes may be candidates for coalescing. The combination of index pre-construction based on distribution prediction and subsequent fine-tuning using node splitting and coalescing has proved to work well in practice, as demonstrated in the following section.

5. Performance Experiments

A series of experiments were carried out to compare the performance of R-Trees, SR-Trees, Skeleton R-Trees, and Skeleton SR-Trees. The SR-Trees reserved 2/3 of the non-leaf node entries for branches to lower level nodes, thus reserving 1/3 of the entries to store spanning index records. The Skeleton Indexes used distribution prediction by computing histograms in two dimensions based on the first 10,000 tuples, plus node splitting and coalescing. The search for nodes to coalesce was triggered

after every 1,000 insertions among the 10 least frequently modified nodes. The node size at the leaf level was 1 Kb, and was doubled at each successive level for all of the index types.

There were six types of input distributions, and for each type, two data sets were used: one containing 100K tuples and one with 200K tuples. In all cases, the domain of input data values was between 0 and 100,000 in two dimensions. The input distribution types are summarized below.

Interval data distributions (Y-values: points; X-values: intervals):

I1. *Uniform Y-value & uniform size distribution.* Y-values: uniformly distributed over [0, 100K]; X-values: interval center-points uniformly distributed over [0, 100K], difference between interval endpoints uniformly distributed over [0, 100].

I2. *Exponential Y-value & uniform size distribution.* Y-values: exponentially distributed with parameter $\beta = 7000$; X-values: same as I1.

I3. *Uniform Y-value & exponential size distribution.* Y-values: same as I1; X-values: interval center-points uniformly distributed over [0, 100K], difference between interval endpoints exponentially distributed with parameter $\beta = 2000$.

I4. *Exponential Y-value & exponential size distribution.* Y-values: same as I2; X-values: same as I3.

Rectangle data distributions (X- and Y-values: intervals):

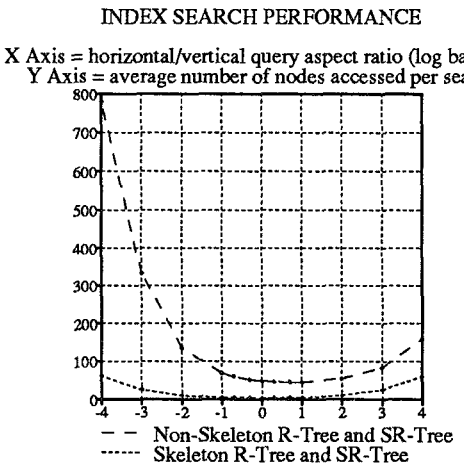
R1. *Uniform size distribution.* Rectangle centroids uniformly distributed over [0, 100K], difference between interval endpoints uniformly distributed over [0, 100].

R2. *Exponential size distribution.* Rectangle centroids uniformly distributed over [0, 100K], difference between interval endpoints exponentially distributed with parameter $\beta = 2000$.

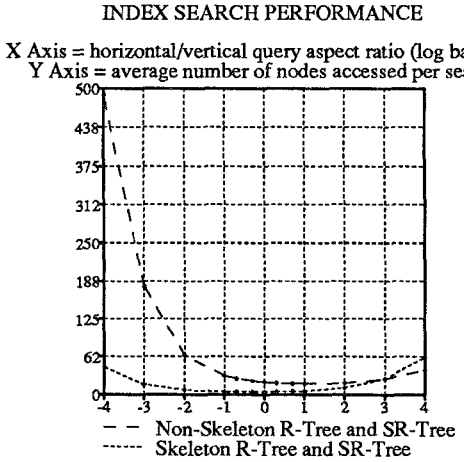
In each experiment, the entire set of data was inserted in random order, and then a number of random searches were performed over the index, where the search argument was a query rectangle of area 1,000,000. The horizontal-to-vertical aspect ratio of the query rectangle (hereafter referred to as the *query aspect ratio*, or QAR) varied over 0.0001, 0.001, 0.01, 0.1, 0.2, 0.5, 1, 2, 5, 10, 100, 1000, and 10000. For each QAR, 100 search rectangles were generated whose centroid was randomly centered over the domain and each was used to perform a search of the index. During each search, the number of index nodes accessed was recorded. Following each set of experiments, for each index type and value of QAR the average number of nodes accessed per search was calculated.

5.1. Results of Performance Experiments

Graphs 1 and 2 show the search performance results (as measured by the number of index nodes accessed) of the four index types on the interval data distributions I1 and I2 containing 200K tuples, respectively. The vertical axes of these graphs plot the average number of index nodes accessed per search, and the horizontal axes plot the log (base 10) of the QAR. In these



Graph 1: Line segment data with uniform length and uniform Y-value distributions



Graph 2: Line segment data with uniform length and exponential Y-value distributions

experiments, both of the non-Skeleton Indexes had identical performance, and the Skeleton Indexes had nearly identical performance.[3] This is because all of the intervals were relatively "short" (uniformly distributed over [0, 100]), and therefore there were very few spanning segments to distinguish the SR-Tree from the R-Tree's performance characteristics. Both of the non-Skeleton Indexes performed much worse than the Skeleton Indexes in the *vertical QAR range* (log of QAR less than zero, hereafter referred to as the VQAR range). This is because the non-Skeleton Indexes exhibited a great deal of horizontal overlap since the data

[3] In cases where the differences between the performance results of some two of the four index types were either zero or so small relative to the scale of the graphs as to be imperceptible, both results were plotted as a single curve.

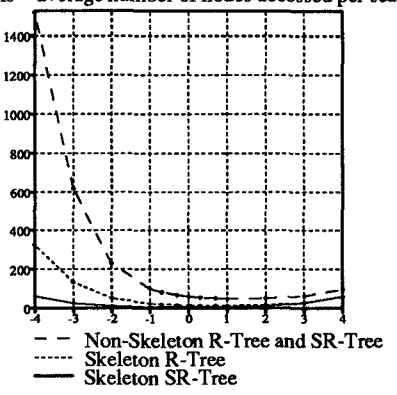
consisted of horizontal segments, whereas the Skeleton Indexes exhibited much less overlap. In Graph 1, in the *horizontal QAR range* (log of QAR greater than zero, hereafter referred to as the HQAR range), the Skeleton Indexes performed better than the non-Skeleton Indexes. In Graph 2, the Skeleton Indexes performed better than the non-Skeleton Indexes up to a QAR of 1,000. In Graph 2 in the HQAR range above 1,000, the non-Skeleton Indexes had a slight advantage. The difference in performance between the Skeleton and non-Skeleton Indexes was much greater in the VQAR range than in the HQAR range. The reason for this and the cross-over effect in Graph 2 is that the non-Skeleton Index non-leaf nodes covered regions that were mostly horizontal, resulting from the preponderance of vertical splits[4] which was a direct result of the type of data being indexed, i.e., horizontal line segments. Most R-Tree (and SR-Tree) node splits were vertical because horizontal line segment data exhibits overlap in the horizontal dimension, but none in the vertical dimension, thus making vertical splits the only viable choice in most cases. This characteristic gave the non-Skeleton Indexes a slight advantage in the HQAR range above 1,000 in the case of exponentially distributed data (Graph 2), but also a large disadvantage in the VQAR range.

The reason why there was a "cross-over" in performance between the non-Skeleton and Skeleton Indexes in Graph 2, but not in Graph 1, was due to the varying extent to which the entire space was covered by overlapping nodes in the case of the non-Skeleton Indexes. In Graph 2, the exponential Y-value distribution caused the Skeleton Index partitions to be "short" at the low end and "tall" at the high end of the vertical dimension, while the non-Skeleton Indexes had a high concentration of very horizontal overlapping non-leaf node regions in the low end of the vertical dimension. For very horizontal queries (QAR > 1,000), the very horizontal, highly overlapping non-leaf nodes of the non-Skeleton Indexes provided slightly better performance than the somewhat less horizontal and mostly non-overlapping regions of the non-leaf nodes of the Skeleton Indexes. The cross-over effect was not present in Graph 1, in which the experiments involved uniformly distributed data values. In that case, since the data was more "spread out", the total area covered by highly overlapping non-leaf nodes in the non-Skeleton Indexes was greater than in the case of the exponentially distributed data experiments in which the overlapping nodes were more concentrated in the lower end of the vertical dimension. Thus, given a uniform distribution of query rectangle centroids, the performance of the non-Skeleton Indexes was degraded due to overlapping nodes to a greater extent in the case of the uniformly distributed data (Graph 1) than in the case of the exponentially distributed data experiments shown in Graph 2.

Graphs 3 and 4 show the results for the exponential interval length distributions. These graphs show that the Skeleton SR-Tree substantially outperformed the Skeleton R-Tree in the VQAR range. This was because there were many spanning segments to distinguish the Skeleton SR-Tree from the Skeleton R-Tree. The Skeleton Indexes only marginally outperformed the non-Skeleton Indexes in the HQAR range in Graph 3 because the mostly horizontal node regions of the non-Skeleton Indexes aided their performance in this range, though they were also generally hampered by overlap. In Graph 4, there is the same cross-over effect as in Graph 2 in the very high HQAR range, and the reason for it is the same as stated in the discussion of Graph 2. The difference in performance between the SR-Tree and R-Tree was very slight in the non-Skeleton Index case[5], since the regions covered by the non-leaf nodes of the non-Skeleton Indexes were mostly horizontal in both cases, thus allowing few spanning segments to be stored in the higher level nodes.

INDEX SEARCH PERFORMANCE

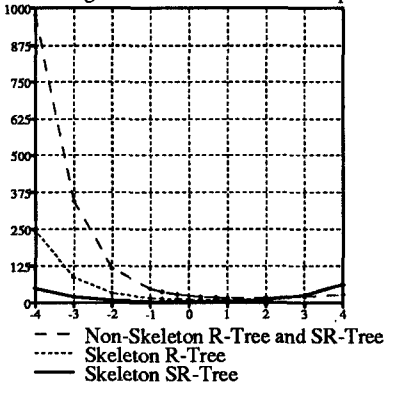
X Axis = horizontal/vertical query aspect ratio (log base 10)
Y Axis = average number of nodes accessed per search



Graph 3: Line segment data with exponential length and uniform Y-value distributions

INDEX SEARCH PERFORMANCE

X Axis = horizontal/vertical query aspect ratio (log base 10)
Y Axis = average number of nodes accessed per search



Graph 4: Line segment data with exponential length and exponential Y-value distributions

[4] A *vertical split* of a rectangle means that the splitting line is perpendicular to the vertical axis.

[5] This difference was too small to represent by plotting separate curves due to the scale of the graphs.

These results show that Skeleton SR-Trees are a good choice for indexing horizontal line segment data, as would be typical of historical data. This structure offered the best overall performance for rectangular queries over a broad QAR range. The Skeleton Indexes outperformed their non-Skeleton counterparts to a great extent in the VQAR range and to a lesser extent in the low HQAR range. The SR-Tree outperformed the R-Tree only in the case of the Skeleton Indexes, particularly in the VQAR range. Non-Skeleton Indexes were only superior to Skeleton Indexes in the high HQAR range when the distribution of the Y-values was non-uniform.

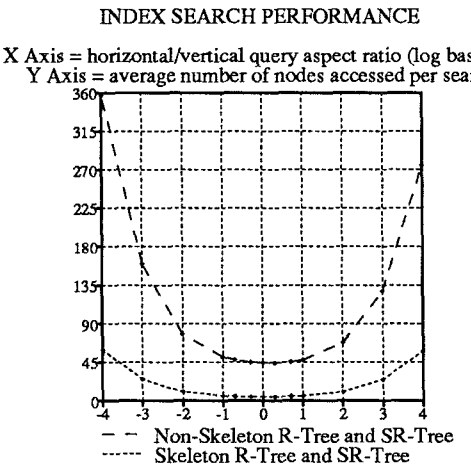
Comparing the results of the experiments that involved exponentially distributed Y-values (Graphs 2 and 4) with those of the uniformly distributed Y-values (Graphs 1 and 3), the Skeleton Indexes using distribution prediction with node splitting and coalescing performed well in both cases. As one would expect, the experiments involving exponentially distributed data always had lower average node accesses per search than the ones involving uniformly distributed data, since the search rectangles were uniformly distributed over the data domain.

Graphs 5 and 6 show the search performance results of the four index types on the rectangle data distributions R1 and R2 containing 200K tuples, respectively. In the experiments whose results are presented in Graph 5, both of the non-Skeleton Indexes had identical performance, and the Skeleton Indexes had nearly identical performance, as was the case in the experiments of Graphs 1 and 2. This is because all of the intervals were relatively "short" (uniformly distributed over [0, 100]), and therefore there were no spanning rectangles to distinguish the SR-Tree from the R-Tree's performance characteristics. In Graph 5, the Skeleton Indexes greatly outperformed the non-Skeleton Indexes, and all of the indexes provided nearly symmetric performance over the QAR range.

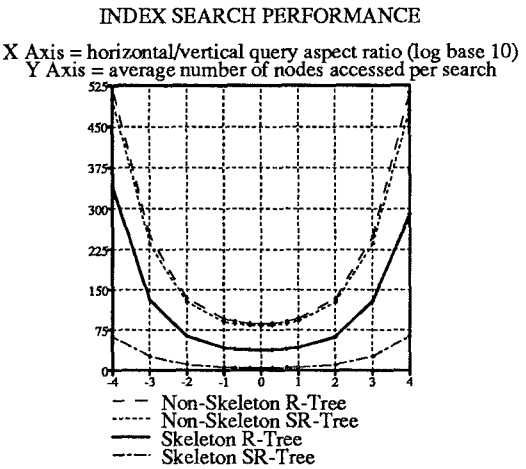
Graph 6 shows the results of the four index types on the rectangle data distribution R2 that featured exponentially distributed interval lengths. This graph clearly shows the superiority of the Skeleton SR-Tree over all of the other three indexes, and the improvement provided by Skeleton R-Trees over the non-Skeleton Indexes. In this set of experiments, large spanning rectangles were stored in non-leaf nodes which provided the Segment SR-Tree with a large performance improvement with respect to the other indexes.

The results of Graphs 5 and 6 show that the Skeleton Indexes are advantageous over non-Skeleton Indexes when indexing rectangle data, and the Skeleton SR-Tree is superior for rectangle data with non-uniform interval length distributions.

The results shown in Graphs 1-6 correspond to the data sets that contained 200K tuples for each of the data distributions I1-I4 and R1-R2. The results of the experiments involving data sets of size 100K for each of the six distributions were qualitatively similar to those in Graphs 1-6, and differed only in that the magnitudes of the results were smaller. Therefore, the results of the experiments involving 100K tuples were omitted in the interest of brevity. Also, experiments involving rectangle data with exponential centroid distributions and both uniform and exponential interval length distributions were performed, and the results



Graph 5: Rectangle data with uniform interval length and uniform centroid distributions



Graph 6: Rectangle data with exponential interval length and uniform centroid distributions

were qualitatively similar to those shown in Graphs 5 and 6, respectively, and were therefore omitted. The results of these and other experiments are included in [KOLO90].

6. Summary and Conclusions

We have introduced the concept of *Segment Indexes* as a novel way of storing multi-dimensional interval data by proposing a set of techniques that may be applied to a class of database indexing structures which are based on paged, balanced, multi-way, tree-structured indexes. Combinations of these techniques were applied to the R-Tree resulting in three indexing structure variations: the *SR-Tree*, *Skeleton R-Tree*, and *Skeleton SR-Tree*. Performance results comparing R-Trees, SR-Trees, Skeleton R-Trees, and Skeleton SR-Trees were presented which demonstrate that Skeleton Segment Indexes provide substantial search

performance improvement over conventional indexing techniques for both rectangle and line segment data. In the case of rectangle data with non-uniform interval length distributions, Skeleton SR-Trees were shown to be superior to all of the other index types considered.

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.

[BECK90] Beckmann, N., Kriegel, H., Schneider, R., Seeger, B. "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles", Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, June 1990.

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

[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.

[KOLO89] Kolovson, C. and Stonebraker, M., "Indexing Techniques for Historical Databases", Proceedings of the Fifth International Conference on Data Engineering, Los Angeles, CA, Feb. 1989.

[KOLO90] Kolovson, C., "Indexing Techniques for Multi-Dimensional Spatial Data and Historical Data in Database Management Systems", PhD Dissertation, Electronics Research Laboratory Memorandum No. UCB/ERL M90/105, University of California, Berkeley, CA, Nov. 1990.

[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.

[STON90] Stonebraker, M., Jhingran, A., Goh, J., Potamianos, S., "On Rules, Procedures, Caching, and Views in Data Base Systems", Proc. 1990 ACM-SIGMOD Conference on Management of Data, Atlantic City, NJ, May 1990.