INDEXING TECHNIQUES FOR MULTI-DIMENSIONAL SPATIAL DATA AND HISTORICAL DATA IN DATABASE MANAGEMENT SYSTEMS

by

Curtis Philip Kolovson

Memorandum No. UCB/ERL M90/105

14 November 1990

INDEXING TECHNIQUES FOR MULTI-DIMENSIONAL SPATIAL DATA AND HISTORICAL DATA IN DATABASE MANAGEMENT SYSTEMS

by

Curtis Philip Kolovson

Memorandum No. UCB/ERL M90/105

14 November 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

DEDICATION

For my parents, June and Melvin Kolovson.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Professor Michael Stonebraker, for having given me the opportunity to work with him, and for providing indispensable guidance for this research. It has been a great privilege to have worked with Professor Stonebraker, as he has taught me how to propose and pursue research in the field of database management, how to write quality technical papers, and how to give good presentations. Beyond these lessons, I have become a better person for the experience of having worked with Professor Stonebraker, as he has consistently maintained a very high standard of excellence for his students as well as for himself. I have the utmost respect for you, Mike, and I am forever grateful for all you have taught me.

I wish to thank the other members of my thesis committee, Professors Raimund Seidel and Arie Segev, for reviewing this dissertation. The other graduate students in the database research group at UC Berkeley, including Spyros Potamianos, Wei Hong, Anant Jhingran, Margo Seltzer, and Mark Sullivan provided helpful comments on this work, as well as their friendship and good humor.

I also wish to thank my colleagues at Hewlett-Packard Laboratories for supporting me throughout my PhD graduate school experience. I especially thank Dan Fishman and Marie-Anne Neimat for supporting my application for the Hewlett-Packard Laboratories Resident Fellowship Program.

Finally, I thank Mihaela for her love and support throughout the final ten months of this endeavor.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1. Overview

Many database applications require the ability to manage large collections of spatial data objects, such as Geographic Information Systems (GIS) and Image Processing Systems [SAME89b]. In order to address this need, many researchers in the field of database management are concerned with the support of large spatial databases and access methods for such databases. While interest in spatial data management has existed for more than a decade [HERO80], a large number of proposals for spatial access methods and GIS systems have only recently appeared [BLAN90, HUTF90, JAGA90a, JAGA90b, OHSA90, SEEG90, SHEN90, GUNT89, HENR89, LOME89a, OOI87, OOI89, SALZ89]. These proposals differ with regard to the type of spatial data that may be indexed, the operations on the data that are supported, and the method used for partitioning the space and for representing the spatial data.

Another important class of database indexing structures involves support for *historical* data. Historical data are maintained by *append-only* database systems which never overwrite or delete previous data values. Research in database management systems that support historical data has also been quite active [SNOD86], and a number of access methods and storage organizations have been suggested for such data [ELMA90, GUNA90, KOLO89, LOME89b, ROTE87, AHN86, LUM85, DADA84].

Spatial data and historical data are closely related because historical data may be represented by a special form of spatial data.

The subject of this thesis is indexing techniques for providing efficient support of spatial data and historical data in database management systems. The areas investigated include indexing techniques in support of the following:

(1) spatial data composed of intervals in multiple dimensions,

(2) non-uniform query distributions, and

(3) very large database archives which may span magnetic and optical disk media.

In the following sections, some requirements of modern database systems that have motivated this research are discussed, and reasons why conventional indexing methods do not adequately support these requirements are presented.

1.1.1. Indexing Spatial Data

Spatial data are characterized by multi-dimensional geometric objects. In the two dimensional case, such a collection of objects may consist of points, lines, rectangles, polygons, circles, or arbitrary curves. In the three dimensional case, spatial objects may also include boxes, spheres, or surfaces. Higher dimensional spatial objects are also possible. The task of indexing spatial objects poses special problems as compared to conventional database access methods which predominately deal with one dimensional point data. The challenge is how to organize an index search tree so that it partitions the data space in such a way that data which does *not* satisfy the query is efficiently pruned out of the search, and data which does satisfy the query is retrieved with as few secondary storage accesses as possible. The special properties of spatial data that create difficulties for access method designers are that spatial data may be multi-dimensional and may be overlapping. Multi-

dimensional data cannot be efficiently indexed by conventional single-attribute indexes such as B-Trees. Overlapping spatial data requires that an indexing technique either partition the data space into disjoint regions and replicate the index records in each region that an object intersects, or else partition the space into (possibly) overlapping regions.

1.1.2. Indexing Historical Data

In this section, it is assumed that historical data may be represented by *step-wise constant* data [SHOS86, SEGE87], i.e., data whose values change only at certain time points, but otherwise remain constant. For example, bank account balances may be classified as step-wise constant since the value of an account remains constant between points in time at which deposits or withdrawals are posted. Step-wise constant data may be represented as step functions in the time domain. An example of such data is illustrated in Figure 1.1 for employee salaries. In Figure 1.1, employee salaries remain constant between raises. In addition, the type of temporal databases that are assumed are *rollback databases*, as defined by [SNOD85]. In a rollback database, records are stamped with the transaction commit time rather than with the *effective* or *valid* time for the information.

The historical data may be represented in a geometric sense by a set of points in $(k - 1)$ dimensions, and an interval in the *time dimension*. In Figure 1.1, $k = 2$ and each data interval specifies a point in the salary domain and an interval in the time domain. As an example of a case where $k = 3$, consider a set of historical data based on employee salaries and job classifications. Such a collection of data may be represented by a point in the salary and job classification domains, and an interval in the time domain. Hereafter, when k-dimensional historical data is referred to, the k specifies the dimensionality of the data, *including* the time dimension.

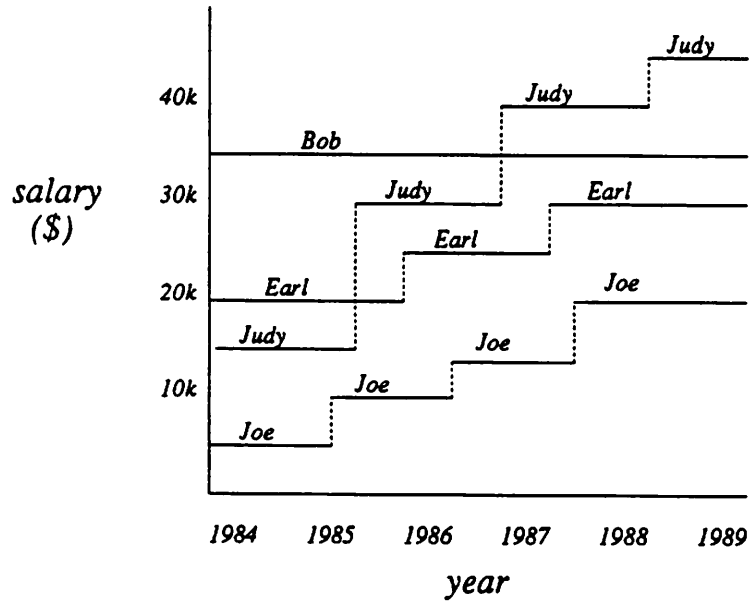


Figure 1.1: Historical data: Employee salaries as a function of time

K-dimensional historical data has special characteristics that are not found in *k*-dimensional *point* data, $k \geq 2$.

(1) Historical data intervals may overlap in the time dimension.

(2) Historical data is generally inserted in an *append-only* fashion, i.e., no deletion of data is performed, and the data are inserted in (approximate) time-sorted order.

(3) The expected query distribution over large historical databases is probably non-uniform, i.e., queries on more *recent* historical data are likely to be more frequent than queries on *older* historical data.

(4) The expected distribution of interval *lengths* of the line segments which represent historical data is likely to be non-uniform. Tuples are inserted into a

historical data relation after having been deleted or updated (replaced) in a current data relation. If deletions and updates of current data tuples are modeled by a Poisson process, the expected interval lengths of the historical data will be distributed exponentially. In that case, there will likely be mostly short intervals and a small number of long intervals.

All of these characteristics must be taken into account when designing new access methods for historical data.

1.1.3. Storing Information about Rules

Database systems that provide a rules system [STON90, HANS90] require some technique to efficiently determine the portion of the database that may be affected by the execution of a rule. In particular, it is highly desirable that rules be processed in such a way so that exhaustive searches of whole relations (or databases) for tuples that are the target of some rule be avoided whenever possible. For example, suppose there is a rule that states that if an employee is paid less or equal to $50,000 he should have a steel desk, and if the employee is paid more than $50,000 he should have a wooden desk. In the above example, two intervals of salary were specified: (0, 50K] and (50K, ∞]. When a new employee is added to the EMP relation, the rules system must determine what sort of desk that employee should have.

In a rules system, there may be a large set of such rules that each may involve an interval of attribute values. The problem arises as to how best represent a large collection of interval data in a search structure so that subsequent point or interval queries may be processed efficiently. In the above example involving rules about employee desk allocation, when a new employee is added to the EMP relation, a potentially large number of salary intervals may need to be examined corresponding to the set of rules that are associated with an employee's salary. In particular, there

may be several such rules in addition to those which deal with desk allocation. This general problem was one motivating influence for the research involving storing interval data in an index that is covered in this thesis.

1.1.4. Problems with Conventional Access Methods

Conventional access methods are not well suited to indexing interval data. Three conventional approaches are considered to illustrate this point. Suppose a B+-Tree [COME79] is used to store intervals using the following straightforward method to insert an interval *I*. First, store index records corresponding to the end-points of *I* in the appropriate leaf nodes. Second, for each interval *I*, store an index record corresponding to *I* with each point data leaf node entry that is contained by *I*. This would clearly lead to an excessively large index, since long intervals would require a large number of redundant index records.

Another approach would be to use an area-based spatial access method, such as the R-Tree [GUTT84] to store the intervals. However, long intervals would create a great deal of inter-node overlap among the non-leaf nodes of the index, thus degrading the search performance. This is due to the R-Tree insertion algorithm which inserts each spatial object into the node that requires the least area expansion to accommodate the object, but nodes are allowed to overlap. An R-Tree index constructed for a collection of short and long intervals would likely have a high degree of overlap resulting from the long intervals. Search performance of such an index would be degraded because R-Trees search all nodes of the index which intersect the given search region. For no-overlap spatial indexes such as the R+-Tree [SELL87], the problem of index size would again be a problem due to the proliferation of redundant index records corresponding to long intervals.

A third approach would be to use a multi-dimensional point data index, such as the K-D-B Tree [ROBI81], to store point data that represents the intervals. For example, the intervals in Figure 1.1 may be mapped into points in 3-space by representing each interval by its 3 coordinates: (x_{low}, x_{high}, y). However, if a *rectangle intersection query* which requests all intervals that overlap a rectangle specified by (rx_{low}, rx_{high}, ry_{low}, ry_{high}), is performed using a 3-dimensional K-D-B Tree, the resulting test for overlap is specified by: ($-\infty < x_{low} < rx_{high}$ and $rx_{low} \leq x_{high} < +\infty$ and $ry_{low} \leq y < ry_{high}$), which involves unbounded range searches in the x_{low} and x_{high} dimensions of the transform space. Clearly such an index will provide poor search performance on such queries.

In spatial databases, it is possible that query distributions may be non-uniform. For example, the most recent versions of a computer-aided design database are likely to be accessed more frequently than older versions. As another example, the most recent historical data (represented as spatial line segments) is likely to be more frequently accessed than older historical data. Although nearly all database indexing schemes work well on both uniform and non-uniformly distributed *data*, all indexes which are based on balanced multi-way search trees assume a uniform *query* distribution. As new spatial data types are included in database systems, a new challenge to database index implementors will be to design indexing techniques that support non-uniform query distributions.

With the rapidly expanding capacity of main memory and magnetic disks, as well as the advent of large capacity optical disks, the sizes of databases and their associated indexes are expected to grow substantially. Existing database indexing methods were designed for magnetic disks, which are rewritable. These techniques do not extend to write-once optical disk media. Database storage managers that include optical disks in their architecture will require new indexing techniques which

may span both magnetic and either write-once or rewritable optical disk media.

The goal of the research presented in this thesis is to address the problems outlined above, i.e., to improve the performance of spatial indexes for multi-dimensional interval data. The next section presents the approaches taken in this research.

1.1.5. Segment, Lop-Sided, and Mixed-Media Indexes

Three major ideas comprise the approaches followed in this thesis: Segment Indexes, Lop-Sided Indexes, and Mixed-Media Indexes. Each is outlined in turn.

The first approach is the exploration of *Segment Indexes*, i.e., indexing structures that are specially adapted to indexing interval data in multiple dimensions, including spatial data. The idea of Segment Indexes is to store each interval in the highest level node of a multi-way, tree-structured index such that the interval spans (covers) the region corresponding to the node, and all of its descendants.

The idea of *Lop-Sided Indexes* is that query distributions which are non-uniform in the indexed domain may be efficiently supported by tree-structured indexes that are not necessarily balanced. To relax the balanced tree criterion so that indexes may become *lop-sided* has not been explored for other than binary trees. In the case of binary search trees, in the case of a static database, an Optimum Search Tree [KNUT73] may be constructed off-line by a dynamic programming technique. For the dynamic case, techniques such as Biased 2-3 Trees [BENT85], D-Trees [MEHL84], and Splay Trees [SLEA85] involve complex restructuring algorithms. The Randomized Search Tree [ARAG89] is an elegant structure which uses a strategy for balancing based on randomization. However, it is not clear how these approaches may be extended to multi-way search trees. The research presented on Lop-Sided Indexes considers the applicability of unbalanced n-ary search trees for n substantially greater than 2.

Mixed-Media Indexes are oriented toward the problem of supporting very large databases in which the associated indexes may span magnetic and optical disk. Several algorithms for dynamically constructing such indexes are presented.

1.2. Outline of the Dissertation

The remainder of this dissertation proceeds as follows. Chapter 2 presents a survey of work related to the subject of this thesis. Chapter 3 describes the concept of Segment Indexes, which are useful for historical data as well as spatial data consisting of arbitrary multi-dimensional interval data. Chapter 4 explores the viability and practicality of Lop-Sided Indexes, i.e., multi-way tree-structured indexes that support non-uniform query distributions. Chapter 5 presents the concept of Mixed-Media Indexes, which are useful for indexing large historical data relations that are maintained in a temporally partitioned storage architecture. Chapter 6 summarizes the results of this work and presents conclusions.

CHAPTER 2

RELATED WORK

2.1. Introduction

This chapter presents a survey of previous work that is related to the research presented in this thesis. Since the subject of this thesis is the indexing of spatial data consisting of multi-dimensional interval data as well as historical data for database management systems, the related work covers at least three areas, as follows.

(1) data structures for intervals that have been developed and studied in the field of Computational Geometry,

(2) spatial database indexing techniques, and

(3) database indexing structures for historical data.

Researchers in each of these areas have contributed important results to the general problem of representing interval data in a data structure to improve the efficiency of exact match (point query) or range (interval intersection) queries over such data. The following three sections survey the work in the three areas mentioned above.

2.1.1. Data Structures from Computational Geometry for Intervals

In the field of Computational Geometry, much work has been done on data structures for representing line segment data that are based on binary search trees [PREP85]. In this section, three such data structures will be briefly described, in order to present the main concepts employed. Since these data structures are based

on binary trees, they are not practical for database indexing structures as they are assumed to be wholly resident in main memory, as opposed to paged onto secondary storage.

In the following discussions, the term *span* is used to refer to the total containment of one interval by another. The definition is made precise below with respect to two intervals, S_1 with endpoints $[l_1, r_1)$ and S_2 with endpoints $[l_2, r_2)$. Interval S_1 is said to *span* interval S_2 if $l_1 < l_2$ and $r_1 \geq r_2$. In all cases, an interval is assumed to be closed on the left endpoint and open on the right endpoint, i.e., an interval whose left and right endpoints are l and r is the interval $[l, r)$. Unless otherwise indicated, the discussion will be limited to the case of a set of intervals in one dimension, and the query of interest is the *one-dimensional interval intersection query*, which for a specified search interval S with endpoints $[l, r)$ retrieves all the intervals that intersect S. Only the case of a static data collection is considered in the discussions of the Segment Tree [BENT77], Interval Tree [EDEL80], and Priority Search Tree [McCR85], i.e., it is assumed that the entire set of data is known and preprocessed before any searches take place. Most database applications involve dynamic collections of data which must support interspersed insert, update, delete, and search operations. While some generalizations of these data structures exist for the dynamic case [EDEL82], they are not considered here in the interest of brevity.

2.1.1.1. Segment Trees

The Segment Tree [BENT77] is a binary search tree which represents a set of intervals in one dimension, or line segments. Its main utility is for finding all the intervals that contain a specified point. To store a set of n line segments in a Segment Tree, the segment endpoints are stored in sorted order in the leaf nodes of a binary tree. The binary tree structure is then built bottom-up as follows. A non-leaf

node of the tree represents the interval whose endpoints are contained in its leftmost and rightmost leaf nodes, respectively. Each segment S_i, $1 \leq i \leq n$, is inserted into the tree as follows. Starting from the root node, if S_i spans the interval represented by the root node, then S_i is added to the linked list of segments which span the root, and the insertion is complete. Otherwise, the process is applied recursively to each of the child nodes of the root, and to each of their children, etc. Each path of the recursive descent is halted when a node is encountered which is spanned by S_i, or which has an empty intersection with S_i.

An example of a Segment Tree is illustrated in Figure 2.2 for the set of line segments shown in Table 2.1 [1].

Line Segment	Left Endpoint	Right Endpoint
A	6	36
B	34	38
C	21	36
D	23	27
E	3	8
F	15	19
G	11	14

Table 2.1: Line Segments used in Examples

In this example and in the examples to follow involving the Interval Tree and Priority Search Tree, each line segment is closed on the left endpoint and open on the right, i.e., line segment A in Table 2.1 is [6, 36). In Figure 2.2, each leaf node is labeled with its corresponding interval number and the leftmost endpoint of the interval, node i corresponds to the interval $[y_i, y_{i+1})$. In addition, nodes are labeled with the

[1] This example was adapted from an example given in [SAME89a], as were the examples of the Interval Tree and Priority Search Tree.

sets of line segments that cover their corresponding intervals. For example, the interval [23, 34) is labeled {A,C} because it is covered by these line segments.

Each line segment is stored at most twice at every level of the tree except at the root level, e.g., a line segment that spans the root node may appear twice at each level below the root. Since the tree has log n levels, the storage required is O(n log n). The time to construct the Segment Tree is O(n log n), since the endpoints are sorted prior to being inserted. The Segment Tree is well-suited to the task of detecting all the segments that contain a specified point, p, since it may be achieved by a single traversal from the root to the leaf node(s) that contain (or bound) p.

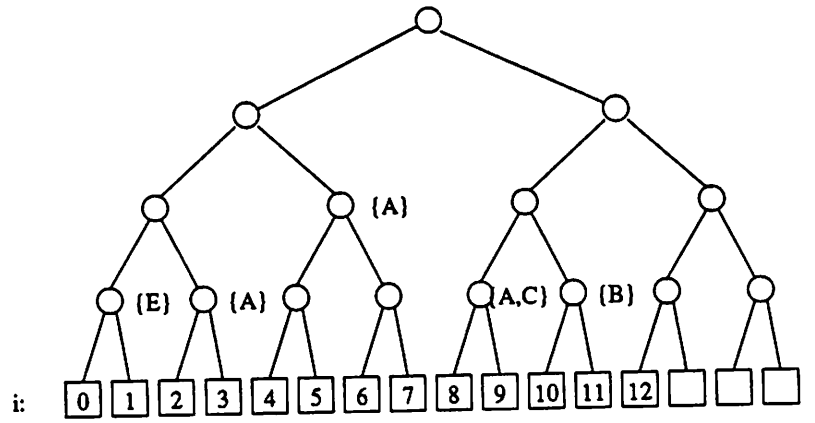


Figure 2.2: The Segment Tree for the data contained in Table 2.1

The linked lists of every node encountered during this search are traversed, thus obtaining the set of segments that contain p. The time to perform a query that searches for all intervals that intersect a specified point is O(log n + R), where R is the number of records returned by the query. The Segment Tree is not so well-suited to the *interval intersection query* that searches for all intervals that intersect a

specified interval S, since to satisfy that query it is necessary to inspect every node in each subtree whose corresponding interval is spanned by S. Thus, to satisfy this query a large portion of the index may need to be searched, and chaining the leaf nodes together would not help since all of the non-leaf nodes whose corresponding intervals intersect S must also be searched.

2.1.1.2. Interval Trees

One of the disadvantages of the Segment Tree is that each line segment may be associated with more than one node of the tree, i.e., an interval *I* is stored with every node whose corresponding interval is spanned by *I*. Also, as mentioned in the previous section, Segment Trees are not well suited to processing the interval intersection query. A data structure which overcomes these problems is the Interval Tree [EDEL80]. The Interval Tree avoids having duplicate entries by associating each line segment S with the *nearest common ancestor* (NCA) of all the intervals that are spanned by S. For line segment S with endpoints [*l*, *r*], the NCA of S is the lowest level non-leaf node that contains *l* and *r* in its left and right subtrees, respectively. An example of an Interval Tree is illustrated in Figure 2.3. In this figure, node 22 is the NCA of line segment A, which corresponds to the interval [6, 36].

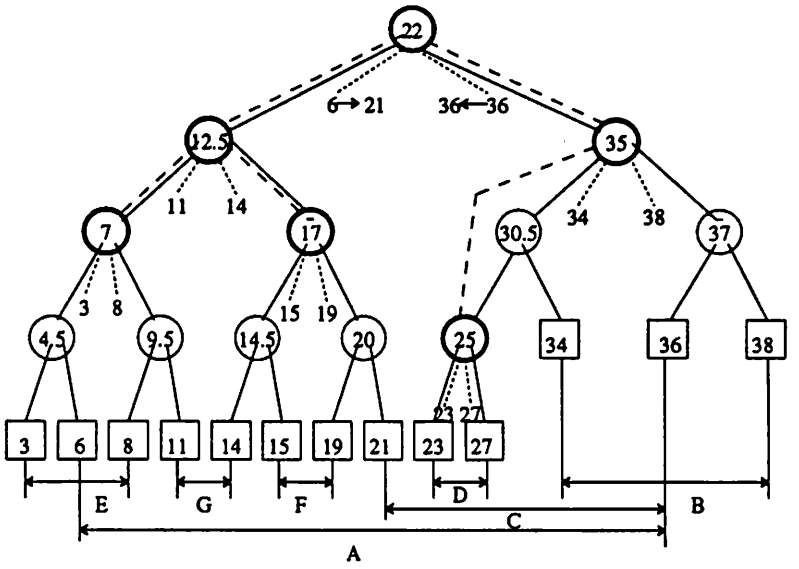


Figure 2.3: Interval Tree for data in Table 2.1

The leaf nodes of the Interval Tree are created in the same way as the leaf nodes of the Segment Tree, i.e., by sorting the endpoints of the line segments and removing duplicates, and assigning each endpoint value to a leaf node. The value assigned to each non-leaf node *N* is the average of the maximum value of its left subtree and the minimum value of its right subtree, and is stored in a field called *Value*(*N*).[2] Each non-leaf node *N* contains a pair of linked lists that contain the sets of left and right endpoints, respectively, of the intervals for which *N* is the NCA. Elements of the left and right endpoint lists are linked in ascending and descending order, respectively. In Figure 2.3, these linked lists are denoted by dotted lines emanating from the non-leaf nodes. For example, node 22 is the NCA of [6, 36) and [21, 36).

[2] *Value*(*N*) may actually be any arbitrary value between these two values, and the average is therefore convenient.

A non-leaf node N is said to be *active* if its corresponding secondary structure is nonempty or both of its child nodes have active descendants. Otherwise, N is *inactive*. Starting from the root, the active nodes are linked in a top-down fashion so that the inactive nodes may be "skipped over" during searches. The active nodes are linked by dashed lines in Figure 2.3, and the active nodes are enclosed by thick circles.

To insert a line segment S with endpoints $[l, r)$, starting from the root, the highest-level non-leaf node N such that $l < Value(N) < r$ is found, and then the endpoints of S are inserted into the linked lists of N. The following is an outline of the algorithm for processing the interval intersection query for a specified search interval S with endpoints $[l, r)$. Starting at the root node, the highest-level non-root node N is found such that $l < Value(N) < r$. Then, starting from N, a search is made for l in the left subtree of N, and similarly a search is made for r in the right subtree of N. At each active node encountered during the search, the linked lists are accessed to produce the endpoints of the intervals that intersect S. This algorithm reports all of the line segments that intersect S, and each intersecting interval is reported once since it is associated with only one node. An example of an Interval Tree search for the interval [l, r) is illustrated in Figure 2.4.

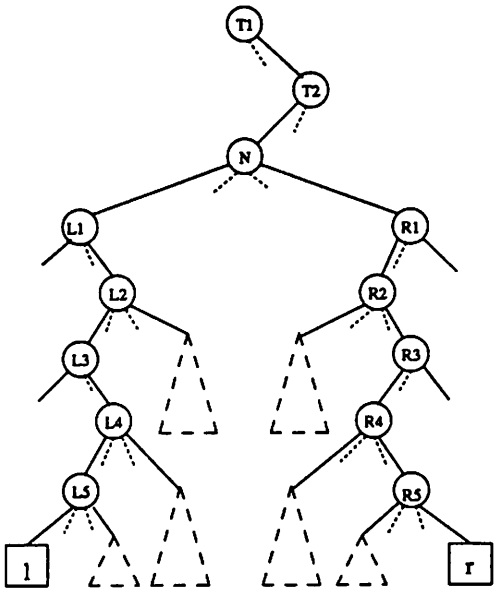


Figure 2.4: Example of an Interval Tree Search

All endpoint linked lists visited are marked with dotted lines, and all of the active interior nodes visited (other than those explicitly shown) are marked by dashed lines.

An Interval Tree requires O(n) space since each interval is stored only once, and may be constructed in O(n log n) time (due to sorting). The time to perform the one-dimensional interval intersection query is O(log n + R), where R is the number of records returned by the query.

2.1.1.3. Priority Search Tree

Another data structure that is useful for satisfying the interval intersection query is the Priority Search Tree [McCR85]. This data structure was previously

referred to as a *treap*, to reflect that it is a hybrid of trees and heaps.[3] A treap is a heap with respect to the y coordinates, and is sorted in the x coordinate in the sense that for a given path from the root to a leaf node L, the contents of all subtrees to the left of the path are less than L and the contents of those to the right of L are greater. The Priority Search Tree is good for performing a two-dimensional *semi-infinite range query*, i.e. to find all spatial objects that intersect the semi-infinite region specified by $[l_x, r_x] \times [l_y, \infty]$. The utility of this structure for the one-dimensional interval intersection query follows from the observation that the interval $[a, b)$ intersects the interval $[c, d)$ if and only if $a < d$ and $c < b$. Equivalently, these two intervals intersect if and only if the *point* (c, d) lies within $[-\infty, b) \times (a, \infty]$. Therefore, to find all one-dimensional intervals that intersect line segment S with endpoints $[l, r)$, the Priority Search Tree may be used to perform the semi-infinite range query $[-\infty, r) \times (l, \infty]$. In a Priority Search Tree, the boundaries of a set of intervals $[l_i, r_i)$ are represented by a corresponding set of points (l_i, r_i).

An example of a Priority Search Tree corresponding to the data in Table 2.1 is illustrated in Figure 2.5. Each line segment $[x, y)$ is treated as a point (x,y) in a two-dimensional space. The leaf nodes contain the x-coordinate values, while the internal nodes contain the maximum y-coordinate values.

---

[3] The term *treap* has since been abandoned by McCreight.

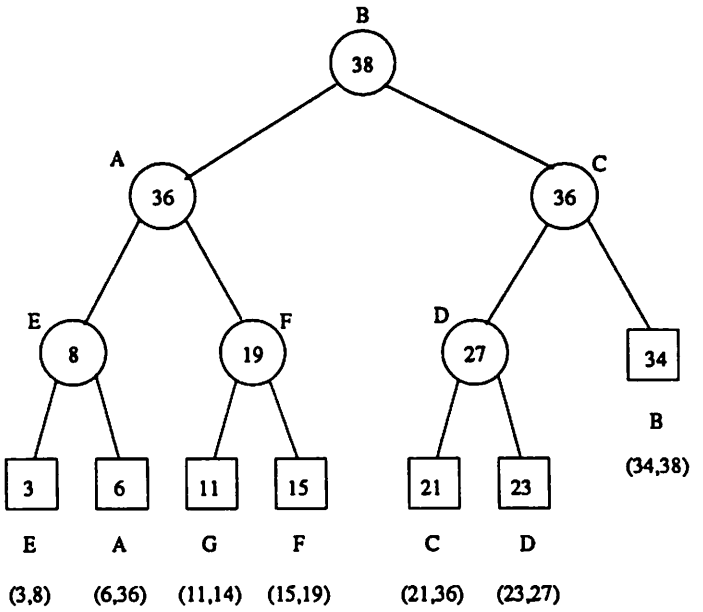


Figure 2.5: Priority Search Tree for the data in Table 2.1

A Priority Search Tree is constructed as follows. First, all of the points (representing the intervals to be stored in the tree) are sorted along their x coordinate, and are then stored in the leaf nodes of a balanced binary tree. Then, starting from the root, the value assigned to each non-leaf node N is the maximum y coordinate in the subtree rooted at N that has not already been stored at a shallower depth in the tree. If no such point exists, the node is left empty.

To perform a semi-infinite range query $[-\infty, r) \times (l, \infty]$ to find all the intervals that intersect line segment $[l, r)$, the algorithm is as follows. Starting from the root, the tree is traversed until the NCA of $[-\infty, r)$ is found at a non-leaf node N. Then, the following algorithm is recursively applied to the subtree rooted at N. Let RS denote the root of the subtree that is currently being searched, and let P_y be the y coordinate of the point associated with RS. If $P_y \geq l$, the following steps are performed:

(1) If $P_x < r$, then P satisfies the query.

(2) If both RS and its right child are on the path from N to $-\infty$, then continue in the right child of RS. Otherwise, if both RS and its left child are on the path from N to r, then continue in the left child of RS. Otherwise, continue in the two child nodes of RS.

If $P_y < l$, the search path is terminated at the subtree rooted at RS since by the heap property P is the point with the maximum y coordinate value in the subtree.

The Priority Search Tree uses $O(n)$ space, and may be built in $O(n \log n)$ time (due to sorting). The time to perform the one-dimensional interval intersection query is $O(\log n + R)$, where R is the number of records returned by the query.

2.1.2. Spatial Database Indexing Techniques for Multi-Dimensional Intervals

In the following sections, techniques for multi-dimensional spatial data indexing are surveyed, specifically with regard to their application for indexing interval data. These techniques include the Hough Transform [JAGA90b], Grid File [NIEV81], K-D-B Tree [ROBI81], R-Tree [GUTT84], R+-Tree [SELL87], R*-Tree [BECK90], Z-Ordering [OREN84], R-File [HUTF90], GBD-Tree [OHSA90], LSD-Tree [HENR89], and the Buddy Tree [SEEG90]. The technique which uses the Hough Transform is special in that it is a technique specifically designed for line segment data where the line segments may have an arbitrary orientation. However, once a certain transformation is applied to the data, it relies on any one of the general-purpose multi-dimensional range indexing methods, such as the other proposals covered in this section. In all of these techniques, line segment data may be stored in one of three ways.[4]

---

[4] This dichotomy is partly due to [SEEG88].

(1) *Ordering technique*: map multi-dimensional coordinates into a lower- dimensional space. For example, map two-dimensional points into one-dimensional points by defining a total ordering on the points that may occur in the original two-dimensional space.

(2) *Transformation technique*: map multi-dimensional objects into points in a higher-dimensional space. For example, map the coordinates that characterize each line segment in $k \geq 1$ dimensions into a point in a $2k$-dimensional space, and store those points in an index for multi-dimensional point data.

(3) *Clipping technique*: divide the space into pairwise disjoint cells, and store with each cell a list of line segments that intersect it.

(4) *Overlapping cells technique*: divide the space into (possibly) overlapping cells, such that each interval is entirely contained by one cell and each cell may contain a set of intervals.

The ordering technique may be applied to any set of multi-dimensional point data in order to transform the data into a set of points in a lower dimensional space. A common case is to define a total ordering on the original two-dimensional point data space so that the (transformed) data has a well-defined one-dimensional ordering. Once this ordering has been established, any conventional one-dimensional point data indexing structure, such as the B+-Tree, may be used to index the (transformed) data. A disadvantage of this approach is that the imposed one-dimensional ordering may be somewhat arbitrary, and data which are in "close" proximity in the original (two-dimensional) space may not be "close" in the (one-dimensional) transform space, and vice versa.

The transformation technique may be applied in any multi-dimensional range indexing structure that indexes point data. Candidate indexing structures for

multi-dimensional point data are the Grid File or K-D-B Tree. As an example of the transformation technique, a one-dimensional interval $[a, b)$ is mapped to a point (a, b). Since $a \leq b$, the region in the transform space that contains points forms a triangle. One disadvantage of this approach is that the point distribution in the transform space is skewed, i.e. only the upper-left triangle of the two-dimensional rectangle that defines the transform space is populated, as illustrated by Figure 2.6 for the data in Table 2.1.[5] Another mapping technique is to map an interval $[a, b)$ into a point (c,d) such that c and d are the centroid (center) and distance from the centroid to the end of the interval (half-length), respectively. An example of this mapping for the intervals in Table 2.1 is illustrated in Figure 2.7.

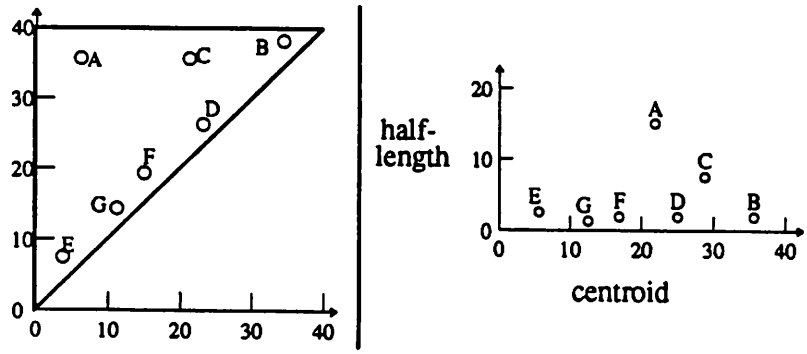


Figure 2.6 (left) and Figure 2.7 (right)

It was suggested by [HINR83] that separating the location parameters from the extension parameters results in a smaller embedding space, which is filled more uniformly. In order to find all intervals that contain a given query point, p, a conic shaped region that is unbounded in the c (centroid) dimension must be searched.

[5] Figures 2.6-2.9 were adapted from [SAME88].

For example, Figure 2.8 illustrates the search region for a point query on point $P = 24$ for the intervals of Table 2.1. All intervals containing P are in the boxed-in region of Figure 2.8. The interval intersection query is handled similarly. Figure 2.9 shows the flat bottomed conic search region that must be searched to find all the intervals in Table 2.1 that intersect the search interval [25, 36). In both of these examples, the search region was unbounded in the d (distance from centroid) dimension.

An example of the third approach, the clipping technique, is the R+-Tree. A drawback of these methods is that information about each line segment has to be stored several times in the index, once for each cell that the line segment intersects. The choice of the cell size is crucial since a smaller cell size provides greater selectivity (resolution), but also has greater storage cost due to replicated information.

The fourth approach, the overlapping cells technique, is typified by the R-Tree [GUTT84]. R-Trees have been shown to be a very good general-purpose multi-dimensional spatial access method [GREE89], are widely used [FROS90, SHEN90, BECK90], and several variants have been proposed [SELL87, BECK90, HUTF90]. R-

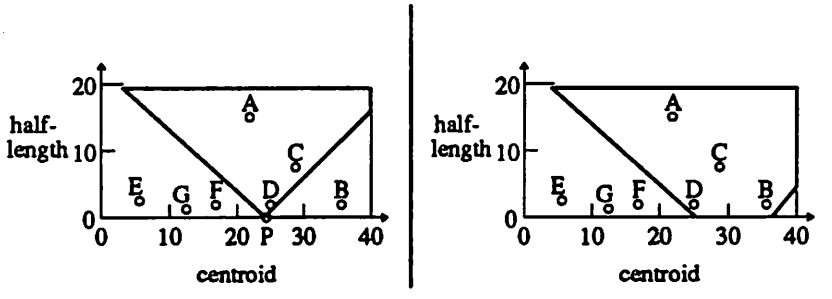


Figure 2.8 (left) and 2.9 (right)

Trees do not partition the entire data space, but only that part of the space that contains spatial objects. The following proposals may be used for indexing line segments using one of the three methods described above.

2.1.2.1. Indexing Line Segments using the Hough Transform

A proposal for indexing line segment data [JAGA90b] involves using a particular transform technique for lines known as the *Hough Transform* [HOUG62]. The Hough Transform of an arbitrarily oriented infinite line segment in two dimensions specified by $y = mx + b$ is simply the *point* (m, b) in the (m, b)*–space*, i.e., a set of ordered pairs whose horizontal coordinate specifies the slope of a line and the vertical coordinate specifies the y-intercept of a line. Finite line segments may be specified by their Hough Transform (m, b) and their projection on the X axis (X_{proj}), i.e., the range x_{min} to x_{max}. Therefore, a set of arbitrarily oriented finite line segments in (x, y) space may be mapped into a set of parallel finite line segments in (m, b, X_{proj}) space. This set of line segments are then indexed by a multi-dimensional range indexing technique, such as one of the other indexing techniques discussed in this section. This approach can be used to satisfy such queries as to find all line segments that (1) pass through a specified point, (2) lie in the vicinity of a specified point, and (3) intersect a specified line segment, and several other generalizations of these queries. This indexing technique is general in that it may be applied to arbitrarily oriented line segment data in multiple dimensions, and was motivated by such diverse applications as machine vision and PC board layout information. The main contribution of this proposal is that it provides a way to transform arbitrarily oriented line segments into a set of parallel line segments. However, this method still relies on a multi-dimensional interval indexing structure for the set of parallel line segments in the transform space.

2.1.2.2. Grid File

The Grid File [NIEV81] is a *flat* directory structured index, as opposed to a tree-structured index, and is useful for indexing point data in k dimensions. Many variations of the Grid File have been proposed, such as EXCELL [TAMM81], the Multi-Level Grid File [WHAN85], the Multi-Layer Grid File [SIX88], and the Generalized Grid File [BLAN90]. With no loss of generality, a Grid File for the case of k equal to two is described. The goal of the Grid File is to perform exact match queries on point data with at most two disk accesses, and to handle range queries efficiently. The Grid File is composed of a *grid directory* consisting of *grid blocks* which are stored in *buckets* (disk pages). There may be one or more grid blocks per bucket, but all the grid blocks in a bucket must form a rectangle.

The grid directory maps grid blocks to buckets, and consists of a *block map* and a set of *linear scales*. The mapping is such that the bucket regions have the shape of a box, i.e., a 2-dimensional rectangle. Several grid blocks may share a bucket, as long as the union of these grid blocks forms a rectangle. The regions of buckets are pairwise disjoint, and together they span the entire space. The block map is a 2-dimensional array containing an entry for each grid block, and each array element may contain a pointer to a bucket. The linear scales are a set of 2 arrays which partition the domain in each dimension. The block map is kept on disk, and the linear scales are kept in main memory. An example showing the retrieval of a record in two disk accesses using a Grid File is illustrated in Figure 2.10.[6] In Figure 2.10, the first search argument is *year* = 1990 and the second is *name* = "Kolovson". The linear scales are used to find the appropriate grid block, and then the block map is used to

---

[6] Figure 2.10 was adapted from [NIEV81].

find the appropriate bucket. In Figure 2.10, the block map entries contain bucket numbers, e.g., bucket number 1 is labeled "B1".

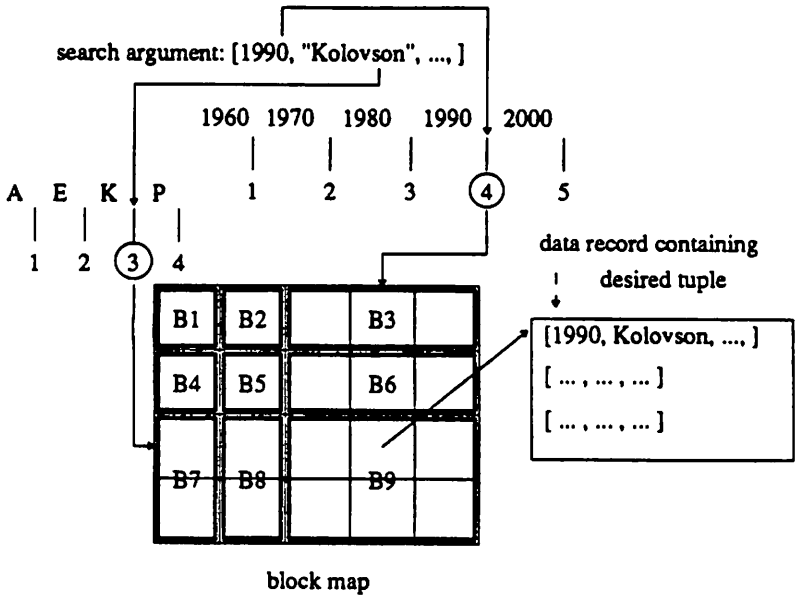


Figure 2.10: Grid File

The first goal of the Grid File is achieved, i.e., any record may be retrieved with two disk accesses: one for the grid block and one for the bucket. Range query performance is highly dependent on the relationship between the grid block partitions and the query interval sizes, as well as to the distribution and correlation of the data. The Grid File is a good indexing structure for uniformly distributed data. However, if the distribution of the multi-attribute keys are skewed, the index can waste large amounts of space. Grid Files were shown to perform poorly on range queries compared to multi-dimensional B-Tree variations when the data was correlated and non-uniformly distributed [KRIE84].

2.1.2.3. K-D-B Tree

The K-D-B Tree [ROBI81] is analogous to a B-Tree, except that each node covers a rectangular sub-region of a multi-dimensional space as opposed to a set of disjoint intervals in a one dimensional space. The K-D-B Tree indexes point data in multiple dimensions, and was one of the first balanced, multi-way, tree-structured indexing structures for multi-dimensional point data that supported exact match and range queries. The K-D-B Tree has many similarities with a B+-Tree: it is a balanced multi-way tree, all data is stored in the leaf nodes, and all non-leaf nodes contain only index records which point to lower level nodes. Also like a B+-Tree, the index adapts to the data inserted by means of node splitting. An example K-D-B Tree is illustrated in Figure 2.11. The dots in the leaf nodes represent the point data records which are contained in the leaf nodes.

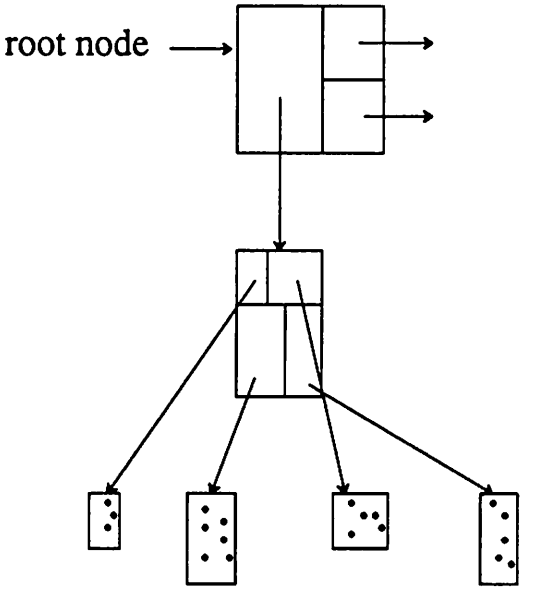


Figure 2.11: K-D-B Tree

The K-D-B Tree assumes that leaf nodes can always be split using only one dimension. There are examples where this assumption does not hold, such as the example of [LOME89a] showing the case where one fourth of the points fall on each half axis (assuming a two-dimensional K-D-B Tree), as illustrated in Figure 2.12(a). Another more frequently occurring problem that arises in K-D-B Trees is that the splitting of a higher level non-leaf node may propagate down to other non-leaf or leaf nodes, as illustrated in Figure 2.12(b). In Figure 2.12(b), if the root node is split vertically (as represented by the dashed line), that vertical split would propagate down to its two children requiring that they be split as well. The problem is that any single plane through the space represented by the index may split the space of one or more descendant nodes. This downward split propagation not only adversely affects the cost of insertions [GREE89], it also may reduce storage utilization [GREE89, SALZ89].

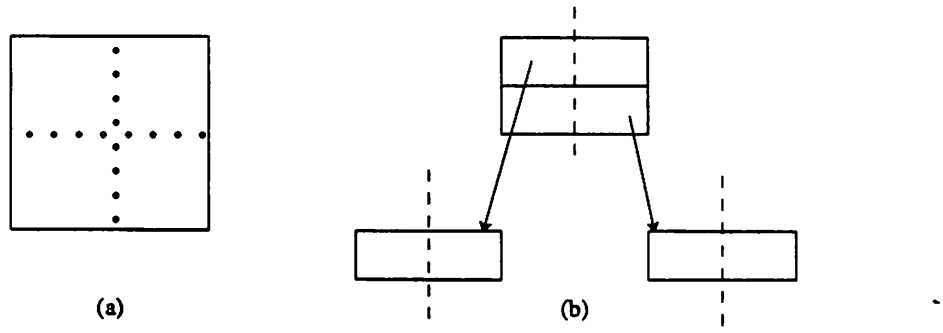


Figure 2.12 (a) and (b): Problems in K-D-B Trees

2.1.2.4. R-Tree

The R-Tree [GUTT84] is a balanced, multi-way, tree-structured index for representing spatial objects using minimally bounding rectangles in k dimensions,

$k \geq 1$. In this discussion, a two-dimensional R-Tree is assumed. All the pointers to data tuples are stored in the leaf nodes, and the non-leaf nodes contain only index records. At each level of the index, the region represented by a node minimally encloses all of its descendants. Objects are stored only once in the index, but more than one search path from the root to a leaf may need to be followed for a given search rectangle, since the regions corresponding to nodes may overlap.

Figures 2.13(a) and 2.13(b) illustrate an example R-Tree structure and the spatial forms it represents.

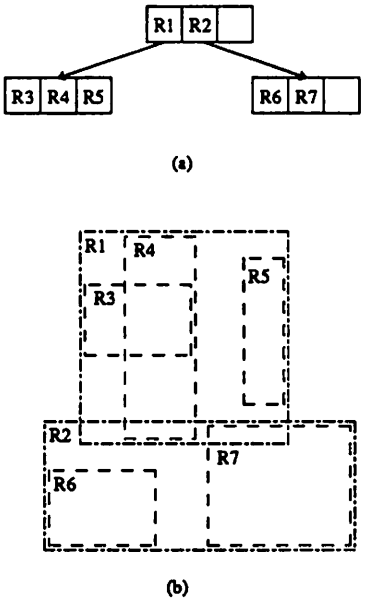


Figure 2.13 (a) and (b): R-Tree

The node entries in Figure 2.13(a) are labeled with the rectangles they correspond to. The geometric representations of those rectangles are shown in Figure 2.13(b). In an R-Tree, the indexed spatial objects (rectangles, in the case of a two dimensional R-Tree) may overlap, and so may the regions covered by the nodes of the index. An

indexed object may intersect or be wholly contained by more than one node.

The R-Tree is based on a heuristic optimization, which is to minimize the area covered by the non-leaf nodes of the index. Minimizing the amount of inter-node overlap is not a goal of the original R-Tree insertion and node-splitting algorithms. The dual goals of minimizing area coverage and node overlap may sometimes be contradictory, as illustrated in Figure 2.14[7], so choosing to minimize area coverage only is a reasonable strategy. Figure 2.14(a) shows four rectangles which must be split between two nodes, i.e., the capacity of one node is three rectangles in this example. Figure 2.14(b) illustrates the splits that would be induced by minimizing the total area covered by the bounding rectangles of the two new nodes. Figure 2.14(c) shows the splits that would be induced by minimizing the overlap between the bounding rectangles of the two new nodes.

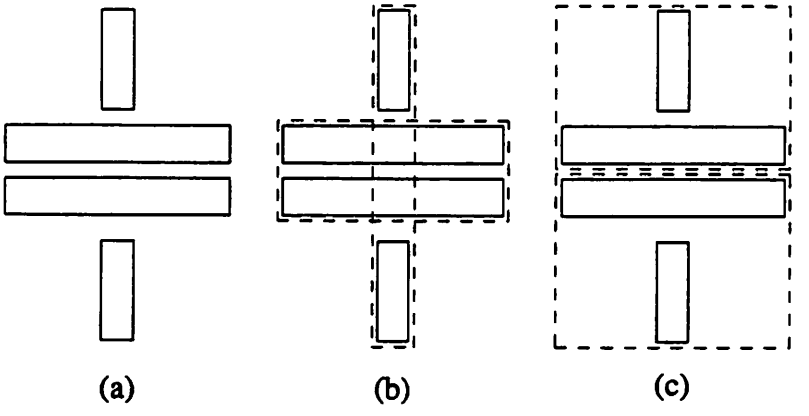


Figure 2.14: Possible splits in an R-Tree

[7] Figure 2.14 was adapted from [SAME88].

The R-Tree has proven to be a very useful spatial data indexing structure, and is very widely referenced in the literature. Recently, it was referred to in [BECK90] as follows:

> "The most popular spatial access method for storing rectangles is the R-Tree... the R-Tree is based on the point access method B+-Tree using the technique of overlapping regions. Thus the R-Tree can be easily implemented which considerably contributes to its popularity."

R-Trees have been successfully used in a large Geographic Information System database project [SHEN90]. The performance of R-Trees was shown to be quite good for a variety of data distributions and query types, as compared with K-D-B Trees, R+-Trees, and 2D-ISAM [GREE89]. Recently, many variations of the R-Tree have been been proposed, including the R*-Tree [BECK90] and the R+-Tree [SELL87]. Among the strengths of the R-Tree are its simplicity and overall good performance. The main drawback to the original R-Tree is that overlapping nodes may hinder search performance for range queries in some cases.

2.1.2.5. R+-Tree

The R+-Tree [SELL87] is similar to both the R-Tree and the K-D-B Tree, and in some aspects is more related to the latter than the former. The R+-Tree proposal was to eliminate overlap among the non-leaf nodes by replicating index records in all the non-leaf nodes whose regions intersect a spatial object. However, R+-Trees require additional space to store the redundant records as compared to an R-Tree. The R+-Tree may be thought of as an extension of K-D-B Trees to cover spatial objects, as opposed to points. An improvement over K-D-B Trees is that nodes of a given level do not necessarily cover the entire space. However, the R+-Tree shares an important disadvantage associated with the K-D-B Tree, namely, the problem of downward split propagation. That is, the splitting of a higher level node may propagate downward to lower level nodes, since overlap is strictly avoided. A

comparison between R-Trees and R+-Trees showed that R+-Trees performed slightly better when the rectangles did not overlap and a small percentage of the space was covered, and that R-Trees performed better when the rectangles overlapped and covered most of the space, but the performance difference between the two indexes was not substantial [GREE89]. In addition, the complexity of an R+-Tree implementation is substantially greater than that of an R-Tree. Also, the R+-Tree has the problem of requiring chaining at the leaf level in order to handle the case when a leaf is full, has an insert request, and all of its current entries overlap. Therefore, R-Trees are generally favored over R+-Trees.

2.1.2.6. R*-Tree

The R*-Tree [BECK90] attempts to use four optimization heuristics simultaneously, namely: (1) minimize the area covered by the non-leaf node regions, (2) minimize the overlap among the non-leaf node regions, (3) minimize the *margin* (the sum of the lengths of the edges of a rectangle) of the non-leaf node regions, and (4) optimize the storage utilization. All of these objectives may not be satisfiable simultaneously. The third optimization objective is particularly noteworthy. For a fixed area, the object with the smallest margin is the square. Therefore, the third objective may be restated as attempting to keep the aspect ratio (the ratio of the horizontal to the vertical edge) as close to 1 as possible. This heuristic may keep the index data clustered in multiple dimensions, and thus could improve the search performance of the index since inter-node overlap would usually be reduced. This approach attempts to avoid the occurrence of node regions which are long in one dimension and short in another.

The changes to the original R-Tree are mainly in the insertion and split routines. In addition, the R*-Tree uses the notion of *forced reinserts* to periodically

redistribute the data in the index. Their strategy is that the first node overflow at each level triggers the reinsertion of p entries from the overflowing node, where p is a parameter (they suggest p = 30% of the node cardinality). This may cause a split in the node which caused the overflow if all entries are reinserted into the same node. Otherwise, splits may occur in one or more other nodes, or no splits may occur.

In their experiments, they compared the R*-Tree to Guttman's original R-Tree for both the linear and quadratic cost split algorithms, and to a variant proposed by Greene [GREE89] for indexing several rectangle data distributions and several query types. Other experiments also compared all of the R-Tree variants to the Grid File for indexing point data. In all cases, the R*-Tree was shown to be superior.

The modifications suggested for the insertion and split routines to attempt to optimize their suggested four criteria appear to be sound and merit further experimental evaluation. The idea of forced reinsertions may be loosely considered as a compromise between the packing algorithm for R-Trees originally proposed in [ROUS85] which is a static method, and the original (dynamic) R-Tree. The packing algorithm of [ROUS85] requires that the entire set of data be inserted at once so that it may be optimally "packed" into an efficient R-Tree. The original R-Tree only performed reinserts when a node became under-full resulting from deletions. The forced reinserts of the R*-Tree incrementally reorganize portions of the index over time, though not guaranteeing any optimal index organization. The authors claimed that the R*-Tree had the best search and insertion performance as compared to all the other R-Tree variants, and stated that the forced reinserts actually improved insert performance by reducing the number of splits.

2.1.2.7. Z-Ordering, or Bit Interleaving

The idea of the Z-Ordering [OREN84][8], or *bit interleaving*, is a simple idea to utilize conventional single attribute indexing structures to index data whose key is composed of more than one attribute, and is an example of the *ordering technique* that maps multi-dimensional points into points in a lower dimensional space. The term Z-Ordering refers to the space filling curve that is traced out by a straightforward application of this method, i.e., bit interleaving using alternating bits in two dimensions, as illustrated in Figure 2.15.

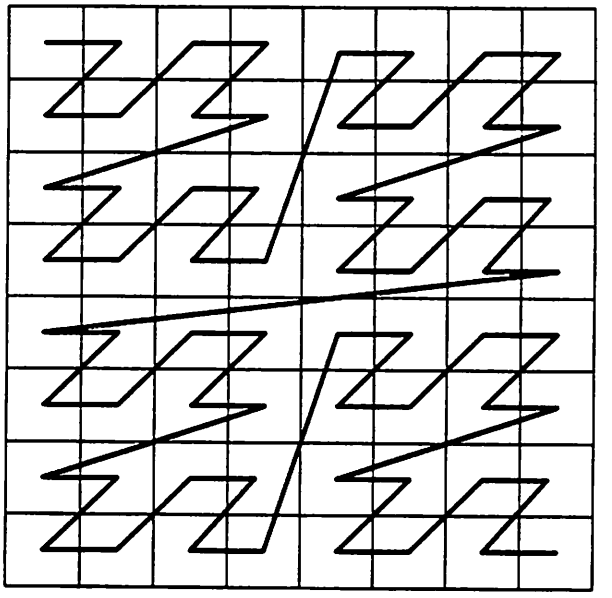


Figure 2.15: Z-Order

The pattern in Figure 2.15 suggests a sequence of Z's. The general technique con-

[8]The concept of the Z-Order is also attributed to [MORT66], and is sometimes referred to as the Morton Order.

sists of transforming a multi-attribute key into one (combined) index key space composed of the interleaved bits of the keys of several attributes, and then indexing this new bit-interleaved key in a conventional one-dimensional index, such as a B+-Tree [COME79]. In this scheme, the method that the bits of the various attributes are interleaved is fixed. The simplest example of bit interleaving of two attributes whose types have equal size (in bits) is to alternate the bits from each attribute, i.e., the odd bit positions in the combined key are taken from the first attribute and the even bit positions come from the second attribute.

The idea of Z-ordering is to impose a one-dimensional total ordering on a multi-dimensional space. Since the decisions that determined the ordering (which are implemented by the chosen interleaving method) may not be optimal over the entire combined key space, or the assumptions that led to the decision may have changed over time, a new ordering may be desired at some point. An example of this is that a certain interleaving method may lead to objects which are in close proximity in the (original) two-dimensional space may be far apart in the (transform) one-dimensional combined key space. In order to change the bit interleaving method of an existing index which uses this approach, all of the combined keys would have to be recomputed and thus the entire index would need to be completely reconstructed. Since it is generally desirable that indexes not require such periodic reorganization, this technique is only of marginal utility as it applies to data whose distributions of all the attributes to be indexed are either known in advance or may be fairly accurately estimated.

2.1.2.8. R-File

The R-File [HUTF90] is another variant of the R-Tree, and its name is somewhat of a misnomer, as it would seem to suggest that it is a closer variant of the Grid File

than the R-Tree, which is not the case. This discussion assumes a two-dimensional R-File for indexing rectangles. The main difference between the R-File and the R-Tree is in the split procedure, and the technique used for encoding the regions that correspond to nodes. The goal of the R-File is to avoid clipping spatial objects so that an object is associated with at most one node, and also to avoid overlap among the node regions. The split procedure of the R-File works as follows. Nodes are successively halved, and the split dimension is alternated in a cyclic fashion. However, rather than creating two nodes which are each half the size of the original node, the two nodes created are *one* of the two halves, plus the original node itself. The half-node chosen is the one that divides the rectangles most evenly between the half-node and the original. A rectangle is stored in the smallest of the nodes within which it lies entirely. Once a node is split between itself and one of its halves, it may subsequently be split between itself and its other half. To illustrate this scheme, Figure 2.16(a) contains 9 rectangles which are to be indexed by both a Grid File (Figure 2.16(b)) and an R-File (Figure 2.16(c))[9].

---

[9] Figure 2.16 was adapted from [HUTF90].

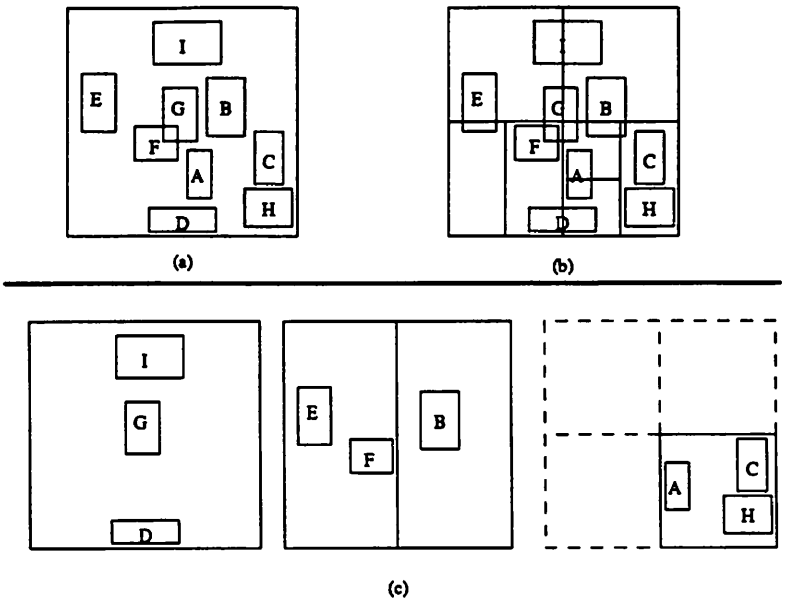


Figure 2.16: 9 rectangles (a) stored in Grid File (b) and in R-File (c)

In both cases, it is assumed that the capacity of a node is 3. Note that in this example, the entire data space node was split between itself and its left half, and later between itself and its right half.

Another departure from the R-Tree is that rectangles and non-leaf node regions are encoded by a pair of numbers representing the level of splitting and the Z-order node number, as opposed to storing the actual rectangle boundaries. For the example of Figure 2.16(c), the nodes that contain rectangles are specified by the following pairs of level and node numbers: 0-0, 1-0, 1-1, and 2-2.

Performance experiments compared the R-File to an R-Tree, and the results showed that the R-File had search performance that was approximately 10%-20% better than the R-Tree. However, primarily due to their region encoding scheme, the fanout of an R-File node was 80 (number of children per non-leaf node) as compared

to 56 for the R-Tree, where both had a page size of 1 Kb. A more fair and meaningful comparison would be to have compared the R-File to the R-Tree using the same page layout, and hence the same branching factor. In addition, only one data set was used in their trials: 48,000 rectangles from a geographic database, consisting of "many very small rectangles and few large ones", and the data set had a *cover quotient*[10] of 5.78. The query regions ranged from 0.25% to 5% of the data space.

Another drawback to the R-File as compared to the R-Tree is that the R-File partitions the entire space, whereas the R-Tree only indexes that part of the data space that is populated. Therefore, for very non-uniform data distributions, the R-File will perform very poorly. This characteristic is something that the R-File shares with the Grid File.

2.1.2.9. GBD-Tree

The GBD-Tree [OHSA90] is another area-based spatial access method, similar to the R-Tree and also to the R-File, and its name stands for Generalized BD-Tree. Unlike the BD-Tree [OHSA83] which is a binary tree, the GBD-Tree is a balanced multi-way tree that stores spatial objects in a hierarchical tree of minimal bounding rectangles, like an R-Tree. In this discussion, a two-dimensional GBD-Tree is assumed. Like the R-File, the GBD-Tree splits node by successive halving and varies the dimension of splitting in a cyclic fashion. Another similarity to the R-File is the use of a special encoding scheme to map regions into index node subdivisions which is similar to the Z-ordering. If a data rectangle overlaps more than one node region, it is stored only in the node that contains the rectangle's centroid. As in the R-Tree,

---

[10] The *cover quotient* is defined as the sum of the rectangle areas divided by the area of the data space, which is an aggregate measure of overlap among the rectangle data.

each non-leaf node entry contains a minimum bounding rectangle which bounds the union of its descendant nodes, and as in an R-Tree these non-leaf node regions may overlap.

The only advantage that the GBD-Tree offers over an R-Tree is that insertions and deletions may be processed more efficiently due to the encoding scheme and because each object is only stored in the node that contains its centroid. These features enable the GBD-Tree to descend a unique path from the root to a leaf for an insertion or deletion of a specified rectangle. However, no apparent advantage is gained with respect to search performance with respect to an R-Tree. The question remains as to whether their cyclic halving splitting scheme will provide good search performance results. Unfortunately, the performance experiments focussed exclusively on storage utilization and a comparison of *insert* performance with the R-Tree. The most important comparison, that of search performance as compared to the R-Tree, was conspicuously omitted. This index also has the drawback of partitioning the entire data space similar to the R-File, in contrast to only indexing the area covered by the data itself, as is done by the R-Tree.

2.1.2.10. LSD-Tree

The LSD-Tree [HENR89] is similar to the K-D-B Tree, except that it is specifically designed to index both multi-dimensional point and spatial objects. It uses the transformation technique to map spatial objects into multi-dimensional points, but the authors claim that they have avoided the pitfalls that have been attributed to using a straightforward application of the transformation technique, i.e., the transform space contains non-uniformly distributed sets of points and often requires a semi-infinite search (one end of a search interval along a dimension is unbounded) in the transform space to process a finite rectangle query. In this section, a 2-

dimensional LSD-Tree is described for storing rectangle data. The transformation technique maps a rectangle into a point in 4-space by choosing the 4 coordinates of the lower left and upper right corner points to represent a 2-dimensional rectangle. Since the lower bound is always less than the upper bound of the side of each rectangle, the points for the rectangle sides only occupy the region on the upper left side of the main diagonal in the transform space, as was illustrated in Figure 2.6.

The LSD-Tree is novel in several respects. It consists of an approximately balanced *binary* tree which is paged onto secondary storage. The paging algorithm ensures that the number of *pages* traversed along any two paths from the root to a leaf node differs by at most one. However, depending on the page size, the height of the balanced binary tree on that page may contain several levels, so the depth of the leaf nodes are guaranteed to be within a range of 0 to k levels, where k is the maximum height of the largest balanced binary tree that may fit on one page.

The index uses either a *data dependent* or a *distribution dependent* split strategy. The former is the conventional splitting algorithm that divides the split node by its median value. The latter chooses the split dimension and value independent of the values of the objects in the node. An example of this strategy is to split a cell into two cells of equal areas, which would be appropriate if the distribution of the objects was uniform. The LSD-Tree stands for *Local Split Decision* Tree, since the decision of how to split a node can be chosen on a locally optimal basis, i.e., optimal with respect to the node to be split and independent from other existing node boundaries. Figure 2.17(a) shows a possible data space partition of a 2-dimensional LSD-Tree, and Figure 2.17(b) shows the LSD-Tree associated with the data space

partition of Figure 2.17(a)[11]. In Figure 2.17(b), each non-leaf node is labeled with the dimension and value of the split that separates its two children, and each leaf node is labeled with the bucket number corresponding to the bucket numbers shown in Figure 2.17(a).

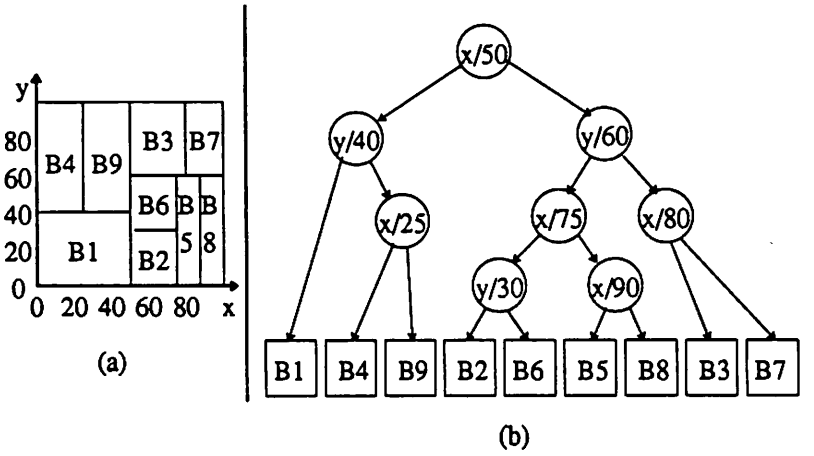


Figure 2.17: (a) partitioned data space associated with (b) LSD-Tree

The distribution dependent split strategy is used to avoid the skewed data distribution that results from the transformation technique. The distribution dependent split strategy adaptively splits the transform space depending on the area of the data rectangles in the index. If most of the data rectangles are small compared to the data space then the transform points tend to be located in a thin strip above the diagonal, and therefore the transform space is split so that the two new nodes contain equally long parts of the diagonal. On the other hand, if the transform points are more uniformly distributed, then the triangular transform space is split into two node such that each one has approximately half the area of the original node. The

[11] Figure 2.17 was adapted from [HENR89].

method used to avoid the unbounded search problem inherent with the transformation technique is to restrict the search in each dimension to the greatest extent of an inserted interval in each dimension.

The authors reported a small number of experimental results which compared the range query performance of the LSD-Tree to the Multi-Layer Grid File [SIX88], which showed the LSD-Tree had a slight advantage.

2.1.2.11. Buddy Tree

The Buddy Tree is described as "a compromise of the R-Tree and the Grid File, but is fundamentally different from each of them" [SEEG90]. The Buddy Tree combines the *buddy system* described in the Grid File paper [NIEV81] with the basic concept of the R-Tree, except that it avoids overlap among the non-leaf nodes, and it also avoids the downward split propagation problem that arose in the K-D-B Tree and R+-Tree. In the (*k*-dimensional) Grid File buddy system, a bucket can merge with exactly one adjacent buddy in each of the *k* dimensions. The assignment of grid blocks to buckets is such that buddies can always merge if the total number of records fits into one bucket.

The Buddy Tree uses the transformation technique of mapping spatial objects into points in a higher dimensional space. For rectangles, the transformation method is to map the 4 coordinates of the lower left and upper right points of the rectangle into a point in 4-space. As pointed out in the discussion of the LSD-Tree, these points (in the transform space) are highly correlated and occupy only a small part of the transform space, yet the Buddy Tree is said to perform well on such distributions.

The index is built by successively halving each node that overflows such that the splitting dimension is chosen in cyclic order. This method is used in order to

maximize the number of possible merge candidates ("buddies") when a merge operation is called for (if a node becomes under-utilized following a sequence of deletions). This method also eliminates the possibility of downward split propagation, since a split value can always be chosen that will not cause lower level nodes to be split as a result of the higher split. For example, Figure 2.18 shows the growth of a Buddy Tree with a node capacity of 4[12]. In Figure 2.18(a), the data node in the upper right region of the directory node has overflowed (the node marked with O has just overflowed). In Figure 2.18 (b), the directory node overflows, and in Figure 2.18 (c) the result is a Buddy Tree of height two. Downward splits are avoided since a node region can always be divided in half, and since all nodes are divided using the halving approach, it is always possible to separate (physically split) a higher level node along a previous (logical) split boundary.

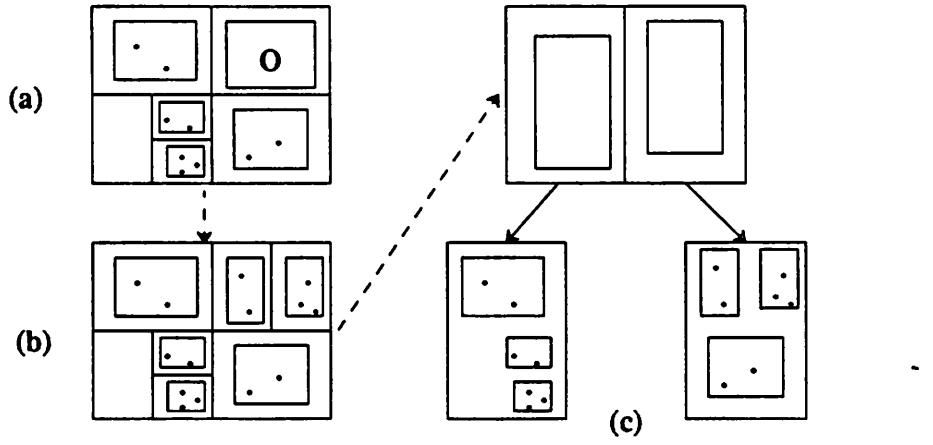


Figure 2.18(a,b,c): Growth of a Buddy Tree

[12] Figure 2.18 was adapted from [SEEG90].

The Buddy Tree uses a technique to increase the fanout of the non-leaf nodes. The idea is to use a *d*-dimensional grid with a dynamically varying resolution for each node. Rectangles are represented by two 2-dimensional points specifying the lower left and upper right corners of the rectangle. Instead of storing the two two-dimensional points as an R-Tree does, each of the points is transformed using a hash function into a single value, so that only two hash values (rather than four coordinates) are stored. The hash function used is the bit-interleaving (Z-ordering) scheme of [OREN84]. This approach increases the non-leaf node fanout substantially, and may be applied to any tree-structured index. In their implementation of a two-dimensional Buddy Tree, the fanout increased by a factor of 3, and the factor would increase for higher dimensions.

Although their performance experiments showed the Buddy-Tree to be generally superior to the Grid File, hB-Tree [LOME89a], and the BANG File [FREE87] for a variety of query types and data distributions, the comparison is somewhat misleading because their node fanout was increased by a factor of 3 using a hashing function, and that same technique was not applied to the other indexing schemes of their comparison study.

2.1.3. Database Indexing Structures for Historical Data

Research in database access methods for indexing historical data has recently become active. The earliest work consisted of very simple ideas, which will only be briefly described. [LUM84] suggested a file organization in which historical tuples are chained in such a way that newer versions point to older versions. This approach would result in extremely bad performance if a query requests data far in the past. [AHN86] suggested several file organizations for historical data, including the *accession list* in which each object has a small directory to access its versions

based on time, which is an improvement over the file organization of [LUM84]. Ahn's approach for secondary indexing was to use a conventional index, such as a B+-Tree, augmented with index record fields containing the endpoints of the intervals for which the indexed tuple was valid. In addition, he limited his consideration to indexes contained only on magnetic disk. The size of such an index would likely become prohibitively large and so would not provide good search performance and would also consume vast amounts of magnetic disk space. For these reasons, his approach would be of limited utility.

The more recent proposals have generally fallen into one of two categories: those that assume that current data are stored separately from historical data, each with their own access methods and (possibly) on different storage media. Such a storage system has been referred to as a *temporally partitioned store* [AHN88]. The motivations for this storage architecture are that:

(1) current data are likely to have a higher query access frequency, and thus access to it should be optimized, and

(2) append-only historical data archives will tend to become quite large, and therefore are most likely to be stored on lower cost optical disks.

The other category that these proposals fall into is that of indexing structures which do not presume a temporally partitioned store. These may be further subdivided into indexes that are to be exclusively contained on optical disks (particularly on write-once optical disks), and indexes that are stored entirely on magnetic disk and assume that the amount of historical data does not exceed some maximum threshold size.

The six proposals surveyed below span the categories outlined above. The Write-Once B-Tree [EAST86] and the Allocation Tree [VITT85] are optical disk only

schemes. The Time Index [ELMA90] assumes that the historical index resides on magnetic disk, and only grows to a maximum threshold size. The Time-Split B-Tree [LOME89b] assumes a temporally partitioned storage architecture. The AP-Tree, Nested ST-Tree, and Nested AT-Tree [GUNA90] may be employed in either an all magnetic disk storage system or one composed of magnetic and optical disks. The Persistent Search Tree [SARN86] may be used in a storage system composed of all magnetic, magnetic and optical, or all optical disks. Each of these proposals are reviewed below.

2.1.3.1. Persistent Search Tree

The Persistent Search Tree [SARN86] was proposed as a useful data structure for the *Planar Point Location Problem*, which is the problem of determining the polygon containing each query point. The Persistent Search Tree is also applicable for indexing historical data since it supports a sequence of *time-stamped* updates. Queries to the structure are made with an extra parameter t denoting time, and the result of such a query is the same as if the query had been made at time t. The Persistent Search Tree is based on the notion of *limited path copying*. The idea of (unlimited) path copying is to copy only the nodes in which changes are made. This implies that all nodes which point to a changed node must also be copied. The effect of path copying is to create a set of search trees, one per update, which have different roots but share common sub-trees. A drawback of path copying is its non-linear space usage. A search tree using the path copying method that contains n records and has had m updates may be searched (for a specified record and time) in time $O(\log m + \log n)$, but requires $O(m \log n)$ space.

Path copying may be eliminated if nodes are allowed to become arbitrarily "fat". Using this approach, changing a pointer in a node is accomplished by simply adding

a new pointer to the node along with the new time stamp. Thus, each node consists of a key and a set of children pointers. A search tree that uses "fat" nodes and no path copying requires only $O(m + n)$ space, but queries take $O((\log m)(\log n))$ time.

The advantages of both of the above methods (the logarithmic query time of path copying and the linear space requirement of the "fat" node technique) may be combined by allowing nodes to grow only up to certain amount, i.e., to use "limited" path copying. The idea is to allow each node to hold k pointers in addition to its original two, where k is some constant. When attempting to add a new pointer to a node, if the node is full, it gets copied, and a pointer to that new node is installed in the parent of the copied node. Therefore, copying propagates up the tree until a node with a free slot is reached, or else the root node is copied. This technique is referred to as the Persistent Search Tree, and it has been shown that this data structure requires $O(m)$ space and that queries take $O(\log m + \log n)$ time [SARN86].

2.1.3.2. Write-Once B-Tree

The Write-Once B-Tree (WOBT) [EAST86] is an indexing structure for historical data which utilizes the path copying technique and was primarily designed for a write-once storage medium, such as a write-once read-many (WORM) optical disk. The WOBT is similar to a B+-Tree, except that index records are augmented with timestamps to indicate when the record was inserted (and hence became valid), and nodes are split either according to their temporal values or data attribute values. The main emphasis of this index is to keep the most recent versions of records in a small number of nodes, enabling search of the current data (or data within a narrow *time window*) to be efficient. The WOBT always performs a *time split*, i.e., the split value is the current time. Sometimes the WOBT simultaneously performs both a *time split* and a *key split* (a *key split* is a split based on a non-temporal attribute

value). Whether a split is by key and current time or by current time alone, only the most recent versions of index entries are copied to the new nodes. An example of a WOBT node split by key and time is illustrated in Figure 2.19(a) [13].

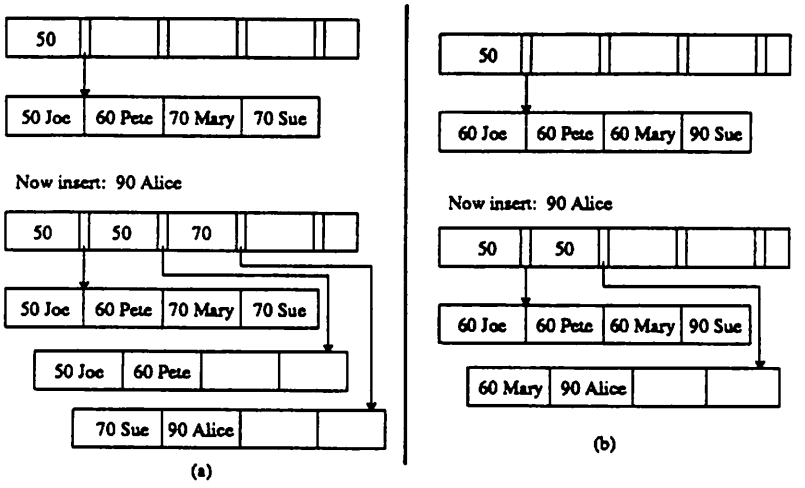


Figure 2.19 (a) and (b): Splitting in a WOBT

The storage medium is assumed to be a WORM optical disk, and thus Figure 2.19(a) shows that the old node remains in the database. If there have been many updates, the number of current versions may be so small that there are not enough current records to make two new nodes. In this case only one new node is constructed, consisting only of the current versions. Figure 2.19(b) illustrates a time split WOBT node. The node splitting policy of the WOBT is summarized as follows.

(1) Always perform a time split, using the current time as the splitting time.

[13] Figure 2.19 as adapted from [LOME89b].

(2) Perform a key split whenever a fraction f or more of the overflowing node consists of current data (f is a parameter)[14].

The policy of time-domain-only splitting accomplishes the desired effect of concentrating the current data in a small number of nodes. However, this also leads to many records having redundant copies in the index. In addition, using a WOBT on a WORM optical disk will waste a great deal of space, since each new entry must use an entire block.

2.1.3.3. Time-Split B-Tree

The Time-Split B-Tree (TSBT) [LOME89b, LOME90] is an adaptation of the WOBT for a temporally partitioned storage architecture [AHN88]. As in the WOBT, index records are augmented with timestamps to indicate the time that the record was inserted (and thus became valid). A relation that is contained in a temporally partitioned store is divided between a *current data relation* and a *historical data relation*, each with its own access methods and (possibly) separate storage media. As records are deleted or updated in the current data relations, they are incrementally migrated to the historical data relations. In the case of the TSBT, migration is done one *node* at a time.

The flexibility offered by the rewritability of the magnetic disk portion of the TSBT allowed changes to be made to the splitting policy of the WOBT in order to improve storage utilization and control the fraction of redundant records. The TSBT differs from the WOBT in that (1) in a time split, nodes may be split using any time *after* the last split time (as opposed to only using the current time as the split value),

---

[14] Values suggested for f are between 1/2 to 3/4 the capacity of a node [EAST86]. [LOME90] used $f = 2/3$ when they compared the WOBT to their Time-Split B-Tree in a simulation study.

and (2) nodes may be split by attribute value and not by time. In the case of time splits, the older versions of records are written into the historical database while the newer versions are kept in the current database. The versions of records that are valid across the split time must be present in both the historical and current databases.

A split entirely by key would be appropriate if the full node contained only new records, i.e., all are the result of insertions and none are the result of updates. Since it would not make sense to make a time split in this case, a split by key is performed.

A time split in a TSBT may be performed using a particular time T as the splitting value. The *Time-Split Rule* is as follows.

(1) All entries with time less than T go in the old node.

(2) All entries with time greater than T go in the new node.

(3) For each key used in some entry, the entry with the largest time less than or equal to T must be in the new node, i.e., the version valid at the split time must be in the new node.

This method of splitting forces some redundancy, as all records which persist through the split time T have copies in both nodes. Two splitting policies were suggested [LOME90], and were compared to the original WOBT splitting policy (WOB Policy), described in the previous section. The *Time-of-Last-Update Policy* (TLU) is as follows:

(1) Always perform a time split unless there is no historical data, and use the time of last update as the splitting time.

(2) Perform a key split whenever two thirds or more of the overflowing node consists of current data.

The TLU policy was chosen because if some insertions are done after the last update, this policy avoids carrying those insertions into the historical node because they do not live across the time split.

The other split policy is the *Isolated-Key-Split Policy* (IKS), as follows:

(1) Perform a time split only when not doing a key split, and use the time of last update as the splitting time.

(2) Perform a key split whenever two thirds or more of the overflowing node consists of current data.

The IKS policy is so called because it tends to perform more (isolate) key splits than the TLU policy, which reduces both the *expansion cost*[15] and the fraction of redundant records in the index.

Much work was done in [LOME90] to analyze the fraction of redundant records, space utilization for single version and multi-version data, and the file expansion cost for the WOB, TLU, and IKS splitting policies. Their analysis assumed a uniform data distribution, and a uniform probability of update (so the length of the historical time intervals was also uniform). These assumptions simplified their analysis, but are not necessarily realistic in practice, and therefore more distributions should be considered, such as exponential.

No analysis or experimental results were presented which measured the *search* performance of this index. A conspicuous omission in their work has been the lack of experimental evidence to show how well this index performs in terms of search performance, and to compare its performance to that of other candidate indexing

---

[15] The expansion cost is defined as the cost per (version of) record added to expand the file, in terms of disk accesses, i.e., the number of disk accesses to insert an index record.

approaches. From the design of this index, it appears to be optimized only for *snapshot queries*, i.e., queries as of a specific time t in the past, or within a narrow time window.

2.1.3.4. AP-Tree, Nested ST-Tree, and Nested AT-Tree

The proposal of [GUNA90] contains a taxonomy of query types on historical data. Four query types are defined: (1) *ST Queries* involve a conjunction of a time-invariant primary key and either a point or interval of time; (2) *AT Queries* involve time-varying attribute values and time; (3) *T Queries* only involve time; and (4) *Multidimensional Queries* which may involve arbitrary conjunctions on relational attributes. Four distinct time specifications are defined: the current time ("now"), an arbitrary point, an arbitrary interval $[t_s, t_e]$, or the whole lifespan of the entity. A time point may also be specified by special symbols T_S or T_E, which specify the beginning or ending times, respectively, of the time interval attributes of tuples in a relation.

Three indexing structures are proposed: the Append-Only Tree (AP-Tree), the Nested ST-Tree, and the Nested AT-Tree. The AP-Tree is similar to a B+-Tree, except that it grows in a left-to-right fashion in order to exploit the time-sorted insert order of historical data. Unlike a B+-Tree the depth of the right-most leaf node may be less than that of the other leaf nodes. Another difference between the AP-Tree and B+-Tree is that the former has forward and backward links between non-leaf nodes and their child nodes as well as between sibling leaf nodes, whereas the latter have only one-way links (from higher to lower level nodes and from left to right siblings, respectively). The AP-Tree is used as a *time subindex* (an index on the time intervals corresponding to tuples) in each of the *nested* indexes, which are described below.

The Nested ST-Tree is proposed for handling ST-Queries where the primary qualification is on the time-invariant primary key, hereafter referred to as the *surrogate*. The Nested ST-Tree consists of a *superindex* that indexes the surrogate, which is organized as a B+-Tree in which each leaf entry has two pointers associated with it: one to the current tuple and one to the root of a time *subindex*. The time subindexes are a set of AP-Trees on the time attributes.

The Nested AT-Tree is proposed for handling AT-Queries, and consists of a B+-Tree superindex on a time-varying non-key (TVN) attribute, and AP-Tree subindexes on the time attributes. However, since the TVN attribute is not a primary (unique) key, tuples that qualify on it are likely to overlap over their associated time intervals. To address this problem, several schemes are suggested for partitioning tuples with a given TVN attribute (or a range of the TVN attribute) over the time dimension in an attempt to minimize overlap. In the Nested AT-Tree, the leaf nodes of the time subindexes contain pointers to buckets that are *pointer pages*, which in turn contain pointers to disk pages. In the event of a pointer page becoming full, overflow pointer pages are chained together. A stated objective is to minimize the amount of pointer page overflow and the duplication of tuple pointers in the pointer page buckets.

The Nested ST-Tree was compared to two alternatives: (1) a B+-Tree indexing a concatenation of the surrogate and the time attributes, and (2) a B+-Tree indexing the surrogate values only where the associated time attributes are stored in a *accession list* pointed to by each leaf entry. Their search performance experiments showed that the second of the two alternatives was the most efficient on the average.

The Nested AT-Tree was not compared to other competing indexing approaches, but experiments were carried out to compare the various time dimension partitioning algorithms. Whereas the time partitioning algorithms are potentially useful, the

Nested AT-Tree would appear to suffer from the extra level of indirection introduced by the pointer pages (between the time subindex and the data tuples), and this inefficiency may be further exacerbated when the pointer pages are chained together as overflows occur. The additional index page accesses introduced by the pointer pages may hinder the performance of this index as compared to other indexing techniques.

2.1.3.5. The Time Index

The Time Index [ELMA90] is a rather simple scheme, and is suggested for a storage architecture in which a single index is used for both current and historical data, and this index is completely contained on magnetic disk. The assumption is made that the amount of historical data contained in the index is never allowed to grow beyond a certain maximum threshold size before it is purged from the database (presumably onto optical disk or magnetic tape, at which point it disappears from the index!). The Time Index approach is to store the endpoints of the historical data time intervals in a standard B+-Tree. To reduce the amount of data stored in each leaf node, the complete list of active intervals is stored only in the first index record on a node, and then subsequent records store incremental changes. For indexing both the time dimension and a non-temporal attribute, the authors suggest a two-level index, e.g., a B+-Tree on the non-temporal attribute, where the leaf node entry of each non-temporal attribute points to a B+-Tree on the temporal attribute. Their performance experiments compared search query performance using the *accession list* file organization (each object has small directory to access its versions based on time) [AHN86], clustering (all versions of an object are clustered on disk blocks), and the Time Index combined with each of these file organizations. They also performed tests using Time Indexes with larger leaf page sizes. Their comparisons showed that

using the Time Index was better than not using the Time Index, which is neither an interesting nor surprising result. The database consisted of 1000 objects, and versions were added based on an exponential distribution for interarrival times. No other information is provided about the size of the database, the mean of the exponential distribution, or the length of the interval queries. It appears that only a small fraction of the database was accessed by each interval query.

The problem with their approach is that if there are a large number of long intervals, the amount of data that would be required to be stored in the leaf nodes would increase substantially, thus making the index very large. Also, the index would become very large in size and yet only a small fraction of the more recent data would likely be accessed. The authors rejected the notion of having separate indexes for the current data relations and historical data relations of a temporally partitioned storage system and claimed that their indexing structure is intended for both current and historical data. However, later they say the following:

> "Because the append-only nature of such a temporal database will eventually lead to a very large file, we assume that a *purge*(t_p) operation is available. This operation purges all versions r with $r.valid\_time.t_e < t_p$ by moving those versions to some form of archival storage, such as optical disk or magnetic tape."

Therefore, what they are actually proposing is an indexing structure for a combined current and historical data relation whose span of history is finite. In fact, they have proposed an index for a current data relation in a temporally partitioned storage architecture and have ignored the task of indexing the historical data relation!

The authors pointed out the special properties of temporal data represented as a sequence of intervals: there is no total ordering of the data, since it is composed of intervals with two endpoints such that the intervals may overlap, a large number of long and short intervals can exist at a particular time point, and the data space is unbounded to the right as opposed to being fixed. They further claimed that a

spatial access method based on regions, such as the R-Tree, are unsuitable for indexing intervals representing historical data because the intervals are likely to have a high degree of overlap which would pose problems for conventional spatial access methods which index data based on minimally enclosing rectangular regions, and the data space is continually expanding to the right so that traditional indexes which assume a fixed data space may not be suitable. However, the authors did not consider modifying spatial access methods to address these problems. Furthermore, their performance experiments merely showed that their index was better than no index, as opposed to some other indexing scheme. A more interesting experiment would have been to compare their approach to one of the spatial access methods, which would have substantiated their out-of-hand rejection that spatial access methods are not useful for indexing historical data.

2.1.3.6. Allocation Tree

The Allocation Tree [VITT85] is an index for a write-once optical disk that was primarily designed for efficiently locating the most recently allocated block on a write-once medium. The author also claimed that it is a suitable indexing structure for historical data. The indexing structure may be classified as a *pointer fill-in method* [RATH84], i.e., it implicitly assumes that disjoint pieces of a write-once optical disk page may be written, so that pointer fields may be filled in after an initial portion of a page has been written. Currently, most WORM optical disks do not provide such a facility.

The basic idea of the Allocation Tree is that it is a linked list of successively taller multi-way trees, where each sub-tree is grown in a top-down fashion. Data records may be stored both in the non-leaf and leaf nodes, and the records are time-stamped so that the index may be searched according to its temporal attributes.

This index was not expressly optimized for a historical data indexing structure, but it does provide the necessary functionality. The author provided no experimental evidence to support the merit of this index. This thesis compares the search performance of this index to an alternative approach in Chapter 5.

2.2. Summary

This chapter has surveyed the previous work that has been done in the field of data structures and database indexing techniques that support the indexing of interval data, spatial data, and historical data. Data structures for indexing interval data have been proposed in the field of Computational Geometry, and most of those proposals are based on binary search trees. Spatial data indexing techniques may be categorized as falling into one or more of the following approaches: the ordering technique, the transformation technique, the clipping technique, and the overlapping cells technique. Historical data indexing methods are based on variations of the path copying technique or on methods involving nested indexing structures.

CHAPTER 3

SEGMENT INDEXES

3.1. Introduction

In this chapter, new techniques are proposed for indexing interval data in K dimensions, $K \geq 1$, 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. In this chapter, the notions of *Segment Indexes* and *Skeleton Indexes* are presented as techniques for indexing multi-dimensional interval data. In addition, a comprehensive set of experimental results are reported which show the performance characteristics of these indexing techniques.

Interval data may be characterized by a set of ordered pairs (l, u) that specify lower and upper bounds in K dimensions, $K \geq 1$, i.e., $[(l_1, u_1), (l_2, u_2), ..., (l_k, u_k)]$. To build efficient structures to index such a collection of data, this work proposes to combine aspects of a memory resident data structure (the Segment Tree [BENT77]) from Computational Geometry with those of a class of 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, aspects of the Segment Tree are merged with features of the R-Tree [GUTT84], and the resulting indexing structure is referred to as the *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, $K > 1$).

This chapter proceeds as follows. Section 2 presents the tactics and motivation for this work. Section 3 describes the algorithms used by the SR-Tree. Section 4 presents the notion of adaptable pre-constructed indexes, which are referred 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 contains a summary and conclusions.

3.2. The Segment Index Approach

The Segment Tree data structure [BENT77] stores line segments in a binary tree by storing the segment endpoints in the leaf nodes, and then associates each segment with the highest level node N that *spans* the values corresponding to the left and right children of N. A segment S_1 is said to *span* another segment S_2 if $S_1.low\_limit \leq S_2.low\_limit$ and $S_1.high\_limit \geq S_2.high\_limit$. The work of this chapter is concerned with adapting Segment Trees to *Segment Indexes*, i.e., to extend the Segment Tree strategy from binary trees in main memory to multi-way trees which are paged onto secondary storage.

The motivation for indexing line segment data in a database system were mentioned in Chapter 1, i.e., to improve the performance of spatial indexing structures, to provide an efficient indexing technique for historical data, and to efficiently index one-dimensional intervals and point data in a single index. This following section presents the tactics used to convert a paged, multi-way, tree-structured database index into a Segment Index.

3.2.1. Tactics

This research proposes 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 distribution, and later made to adapt to the actual input data.

Each of these tactics are further explained in the following sections.

3.2.1.1. 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 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. As explained in Chapter 2, R-Trees allow overlap between node regions and do not partition the input data, whereas R+-Trees disallow overlap among the node regions but do partition the input data if the data intersects more than one node region. If R-Trees were able to store "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. Similarly, if R+-Trees were allowed to store "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.

3.2.1.2. Varying the Index Page 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.

3.2.1.3. Skeleton Indexes

A *Skeleton Index* is an adaptable pre-allocated indexing structure 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 of this chapter.

3.3. 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, the design of one Segment Index is presented 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 or spatial data.

3.3.1. SR-Tree

The *SR-Tree* is defined as the Segment Index adaptation of the R-Tree index. In this section, the insertion, node splitting, and search algorithms utilized by the SR-Tree are described. The algorithm descriptions given below are extensions to the original R-Tree algorithms as presented in [GUTT84], which are not repeated here. With no loss of generality a two-dimensional SR-Tree ($K = 2$) is described. Extensions to the cases of $K = 1$ or $K > 2$ are straightforward.

3.3.1.1. 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 segment, then R is 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 3.1.

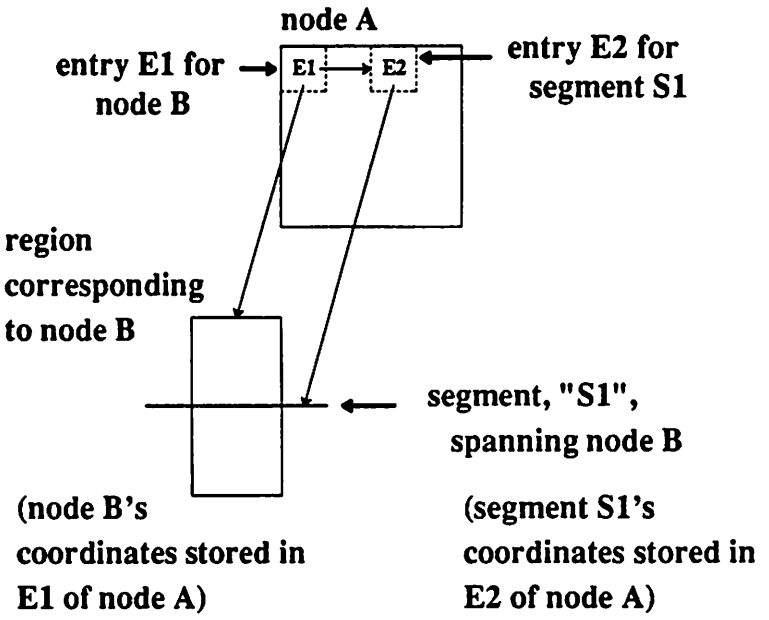


Figure 3.1: SR-Tree storing a spanning segment

In this figure, node A contains a branch index record labeled $E1$ which contains a pointer to child node B. Since segment $S1$ spans the region covered by node B, but not that of node A, $S1$ is stored in 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 using the same insertion algorithm that was applied to the

original record.

An example of a segment cut into spanning and remnant portions is illustrated in Figure 3.2. 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 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 a single (logical) index record. However, these disadvantages are usually not significant because (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 index records.

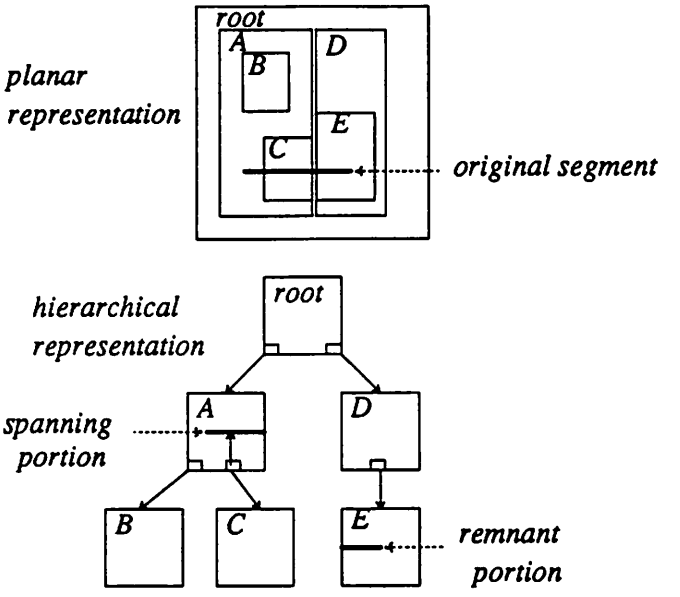


Figure 3.2: Cutting a segment into *spanning* and *remnant* portions

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. That is, if a node was recently expanded to accommodate a new object, that expansion may have increased the region of the node in such a way that intervals which previously had spanned the node before the expansion no longer do after the expansion. 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.

3.3.1.2. 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 3.3.

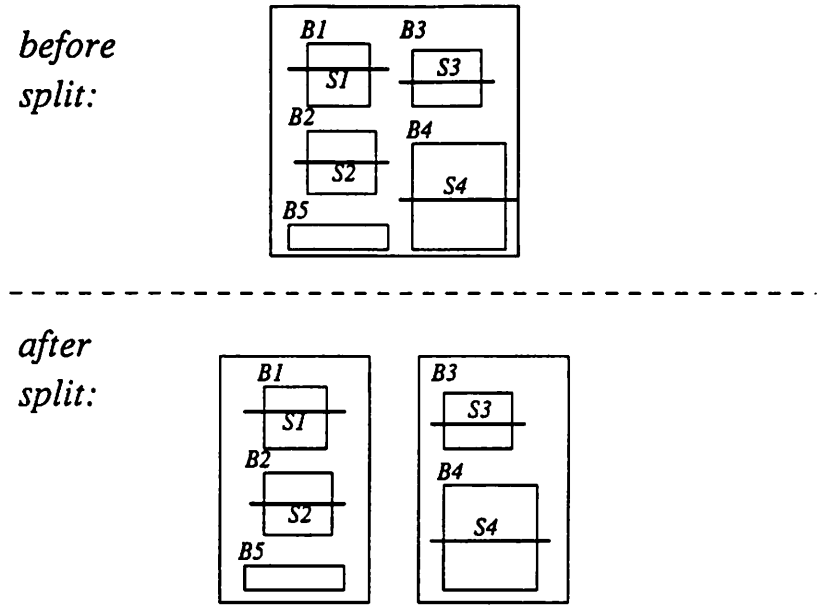


Figure 3.3: Splitting a non-leaf node in an SR-Tree

The top of Figure 3.3 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 3.3 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.

3.3.1.3. Search Algorithm

The SR-Tree search algorithm is similar to that of the original R-Tree. The search algorithm is passed a search rectangle *S*, and the query consists of finding all rectangles that intersect *S*. The algorithm 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.

3.4. 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 *shapes* (i.e., 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.*

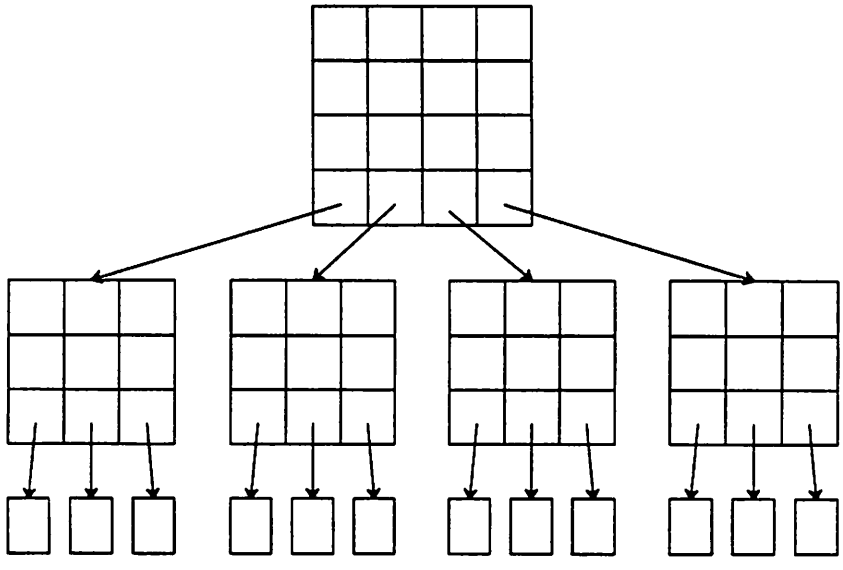


Figure 3.4: A *Skeleton* SR-Tree

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 should be of equal size at each level of the index, as illustrated in Figure 3.4. 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, 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*, 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;
  /* number_of_tuples is the expected... */
  /* ...number of tuples to be inserted */
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;
```

In the calculation above, the reason why the number of nodes at each level is rounded up to be a number whose square root is integral is so that the index may be initially constructed with an equal number of partitions in each dimension, primarily

for simplicity. Therefore, the number of partitions in each dimension at level i is the square root of the number of nodes at level i.[2] 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 3.5.

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.

---

[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.

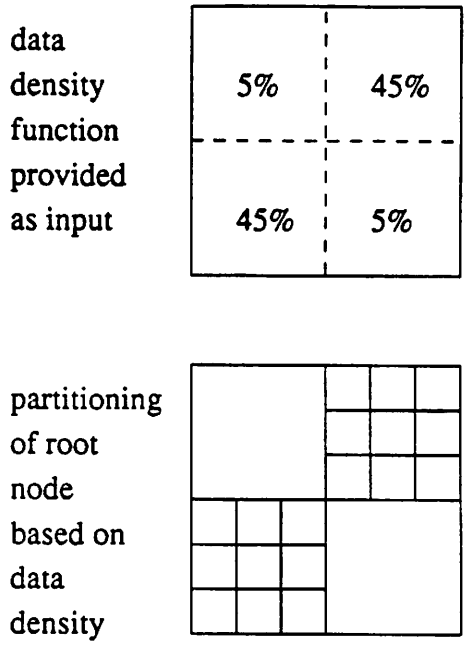


Figure 3.5: Partitioning a Skeleton SR-Tree root node based on a given non-uniform distribution

An alternative approach to starting with a uniform Skeleton Index is to use a technique which will be referred to as *distribution prediction*. 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 the performance experiments described in the next section, values of T as small as 5-10% of the expected number of tuples inserted worked well.[3]

[3] Since the input data in the experiments were generated using random number generators, it is to be expected that distribution prediction would be highly effective in those cases.

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 updated non-empty 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.

3.5. 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 updated non-empty 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 eight types of input distributions, and for each type, two data sets were used: one containing 10^5 tuples and one with 2×10^5 tuples. In all cases, the

range of input data values was between 0 and 10^5 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, 10^5]; X-values: interval center-points uniformly distributed over [0, 10^5], 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, 10^5], 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 center-point* and *uniform size distribution.* X- and Y-values: interval center-points uniformly distributed over [0, 10^5], difference between interval endpoints uniformly distributed over [0, 100].

R2. *Exponential center-point & uniform size distribution.* Y-values: interval center-points distributed exponentially with parameter $\beta = 7000$; X-values: interval center-points uniformly distributed over [0, 10^5]; X- and Y-values: difference between interval endpoints uniformly distributed over [0, 100].

R3. *Uniform center-point* and *exponential size distribution.* X- and Y-values: interval center-points uniformly distributed over [0, 10^5], difference between interval endpoints exponentially distributed with parameter $\beta = 2000$.

R4. *Exponential center-point & exponential size distribution.* Y-values: interval center-points distributed exponentially with parameter $\beta = 7000$; X-values: interval center-points uniformly distributed over $[0, 10^5]$; X- and Y-values: difference between interval endpoints exponentially distributed with parameter $\beta = 200$.

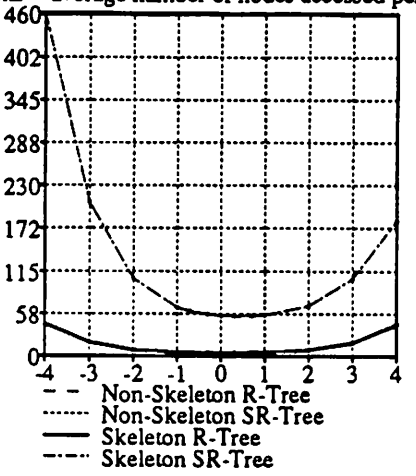
In each experiment, the entire set of data was inserted in random order, and then a number of random searches were performed over the index, where the search argument was a query rectangle of area 1,000,000. The horizontal-to-vertical aspect ratio of the query rectangle (hereafter referred to as the *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 random search rectangles were generated whose center-points were uniformly 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.

3.5.1. Results of Performance Experiments

Graphs 3.1 and 3.2 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 10^5 tuples, respectively.

SEARCH PERFORMANCE OF 4 INDEX TYPES FOR UNIFORMLY DISTRIBUTED DATA (100,000 TUPLES) AS A FUNCTION OF THE QUERY ASPECT RATIO

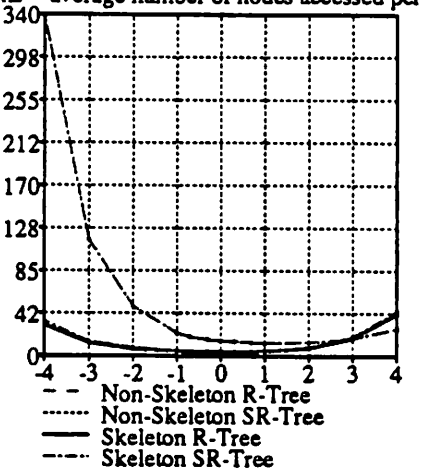
X Axis = horizontal/vertical query aspect ratio (log base 10)
Y Axis = average number of nodes accessed per search



Graph 3.1: Line segment data; Interval length distribution: Uniform [0, 100]

SEARCH PERFORMANCE OF 4 INDEX TYPES FOR 100,000 EXPONENTIALLY DISTRIBUTED (IN VERTICAL DIMENSION) TUPLES, AS A FUNCTION OF THE QUERY ASPECT RATIO

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



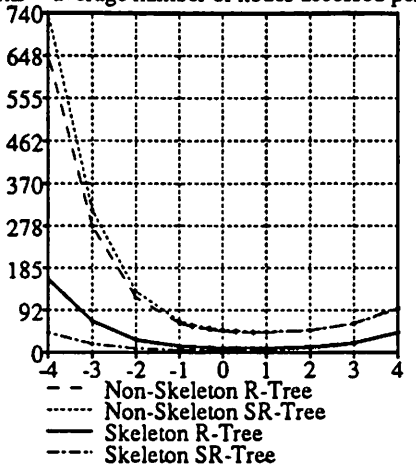
Graph 3.2: Line segment data; Interval length distribution: Uniform [0, 100]

The vertical axes of these graphs plot the average number of index nodes accessed per search, and the horizontal axes plot the logarithm (base 10) of the QAR. In these graphs, both of the non-Skeleton Indexes had identical performance, and the Skeleton Indexes had nearly identical performance. This is because all of the intervals were relatively "short" (uniformly distributed over [0, 100]), and therefore there were no spanning segments to differentiate 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 consisted of horizontal segments, whereas the Skeleton Indexes exhibited much less overlap. In Graph 3.1, 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 3.2, the Skeleton Indexes performed better than the non-Skeleton Indexes up to a QAR of 1,000. In Graph 3.2 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 3.2 is that the non-Skeleton Index non-leaf nodes covered regions that were mostly horizontal, resulting from the preponderance of vertical splits 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 3.2), 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 3.2, but not in Graph 3.1, is 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 3.2, the exponential Y-value distribution caused the Skeleton Index partitions to be "short" at the low end and "long" at the high end of the vertical dimension, while the non-Skeleton Indexes had a high concentration of mostly horizontal overlapping non-leaf node regions in the low end of the vertical dimension. For very horizontal queries (QAR > 1,000), the mostly 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 3.1, 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.

SEARCH PERFORMANCE OF 4 INDEX TYPES FOR
UNIFORMLY DISTRIBUTED DATA (100,000 TUPLES)
AS A FUNCTION OF THE QUERY ASPECT RATIO

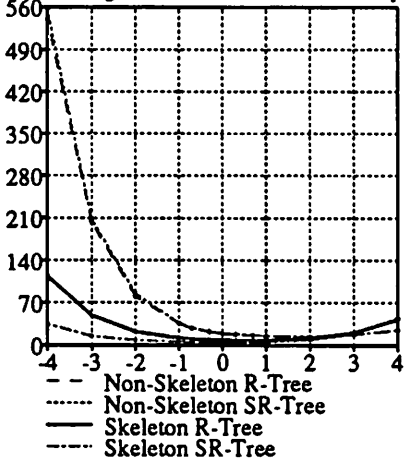
X Axis = horizontal/vertical query aspect ratio (log base 10)
Y Axis = average number of nodes accessed per search



Graph 3.3: Line segment data:
Interval length distribution: Exponential(2000)

SEARCH PERFORMANCE OF 4 INDEX TYPES FOR 100,000
EXPONENTIALLY DISTRIBUTED (IN VERTICAL DIMENSION)
TUPLES, AS A FUNCTION OF THE QUERY ASPECT RATIO

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



Graph 3.4: Line segment data:
Interval length distribution: Exponential(2000)

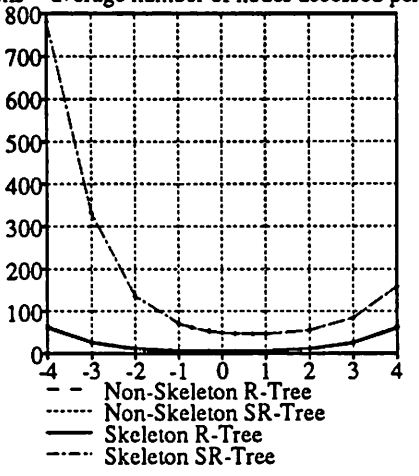
Thus, given a uniform distribution of query rectangle center-points, the performance of the non-Skeleton Indexes was degraded due to overlapping nodes to a greater extent in the case of the uniformly distributed data (Graph 3.1) than in the case of the exponentially distributed data experiments shown in Graph 3.2.

Graphs 3.3 and 3.4 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 differentiate 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 3.3 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 3.4, there is the same cross-over effect as in Graph 3.2 in the very high HQAR range, and the reason for it is the same as stated in the discussion of Graph 3.2. The difference in performance between the SR-Tree and R-Tree was only slight in the non-Skeleton Index case, 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.

Graphs 3.5-3.8 correspond to the same input distribution types as those in Graphs 3.1-3.4, except that the number of tuples was 2×10^5 instead of 10^5. Graphs 3.5-3.8 differ from Graphs 3.1-3.4 principally in the magnitude of the results.

SEARCH PERFORMANCE OF 4 INDEX TYPES FOR
UNIFORMLY DISTRIBUTED DATA (200,000 TUPLES)
AS A FUNCTION OF THE QUERY ASPECT RATIO

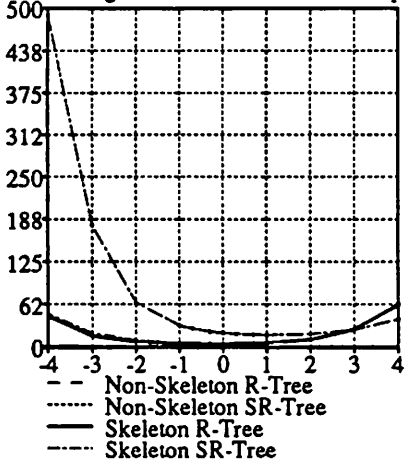
X Axis = horizontal/vertical query aspect ratio (log base 10)
Y Axis = average number of nodes accessed per search



Graph 3.5: Line segment data;
Interval length distribution: Uniform [0, 100]

SEARCH PERFORMANCE OF 4 INDEX TYPES FOR
EXPONENTIALLY DISTRIBUTED DATA (200,000 TUPLES)
AS A FUNCTION OF THE QUERY ASPECT RATIO

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



Graph 3.6: Line segment data;
Interval length distribution: Uniform [0, 100]

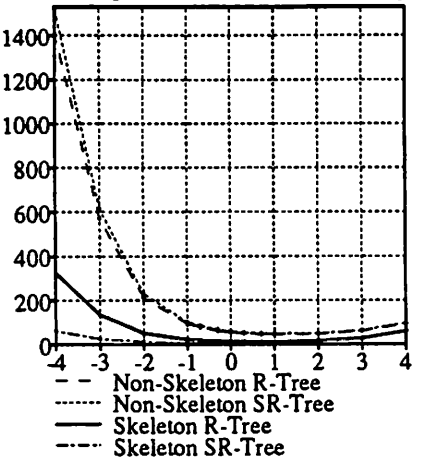
The results of Graphs 3.1-3.8 show that Skeleton SR-Trees are a good choice for indexing horizontal line segment data, as would be typical of historical data. This structure offered the best overall performance for rectangular queries over a broad QAR range. The Skeleton Indexes outperformed their non-Skeleton 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 3.2 and 3.4) with those of the uniformly distributed Y-values (Graphs 3.1 and 3.3), 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 3.9 and 3.10 show the search performance results of the four index types on the rectangle data distributions R1 and R2 containing 10^5 tuples, respectively.

SEARCH PERFORMANCE OF 4 INDEX TYPES FOR
UNIFORMLY DISTRIBUTED DATA (200,000 TUPLES)
AS A FUNCTION OF THE QUERY ASPECT RATIO

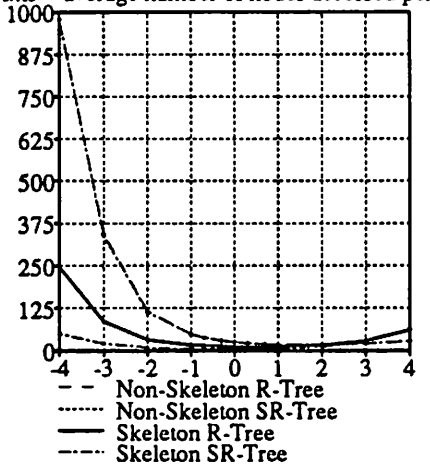
X Axis = horizontal/vertical query aspect ratio (log base 10)
Y Axis = average number of nodes accessed per search



Graph 3.7: Line segment data;
Interval length distribution: Exponential(2000)

SEARCH PERFORMANCE OF 4 INDEX TYPES FOR
EXPONENTIALLY DISTRIBUTED DATA (200,000 TUPLES)
AS A FUNCTION OF THE QUERY ASPECT RATIO

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



Graph 3.8: Line segment data;
Interval length distribution: Exponential(2000)

In Graph 3.9, both of the non-Skeleton Indexes had identical performance, and the Skeleton Indexes had nearly identical performance, as was the case in Graphs 3.1 and 3.2. This is because all of the intervals were relatively "short" (uniformly distributed over [0, 100]), and therefore there were no spanning rectangles to differentiate the SR-Tree from the R-Tree's performance characteristics. In Graph 3.9, the Skeleton Indexes greatly outperformed the non-Skeleton Indexes, and all of the indexes provided nearly symmetric performance over the QAR range.

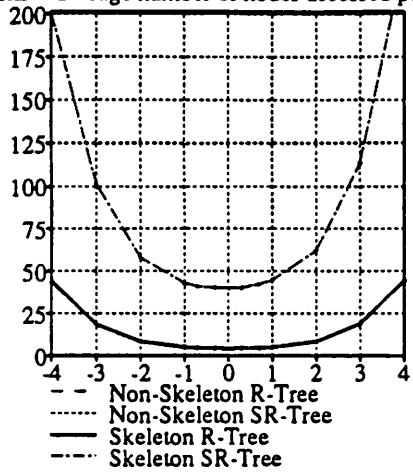
Graph 3.10 shows the results of the four index types on the rectangle data distribution R2 that featured interval *center-points* that were exponentially distributed in the vertical dimension, and an interval length distribution that was uniform over [0, 100]. This graph again shows that the Skeleton Indexes out-performed the non-Skeleton Indexes, and the Segment Indexes performed virtually the same as the non-Segment Indexes. In this graph, both the non-Skeleton and Skeleton Indexes had performance curves that were slightly asymmetrical. An interesting point is that the asymmetry was reversed in each of the two cases, i.e., the non-Skeleton Indexes performed better in the HQAR range, and the Skeleton Indexes performed better in the VQAR range. These asymmetric results were due to the input distribution consisting of center-points distributed exponentially in the vertical dimension, and uniformly in the horizontal dimension. This distribution resulted in non-Skeleton Indexes that favored HQAR queries over VQAR queries, similar to the results shown in Graphs 3.1-3.8. The reason for these results are that most of the non-leaf nodes were mainly horizontal, which is analogous to the experiments involving line segment data with exponentially distributed Y-values.

The asymmetry of the Skeleton Index results, on the other hand, was due to the reduced fanout among the non-leaf nodes in the lower end of the vertical dimension, which resulted from the majority of spanning rectangles being concentrated in these

nodes. This caused HQAR queries to perform worse, particularly those in the lower vertical half of the search space. This "reversed asymmetry" was also present in the experiments involving the line segment data with exponentially distributed Y-values (Graphs 3.2 and 3.6), but was less noticeable because the poor performance of the non-Skeleton Indexes in the VQAR range was the dominant feature of those graphs.

SEARCH PERFORMANCE OF 4 INDEX TYPES FOR
UNIFORMLY DISTRIBUTED DATA (100,000 TUPLES)
AS A FUNCTION OF THE QUERY ASPECT RATIO

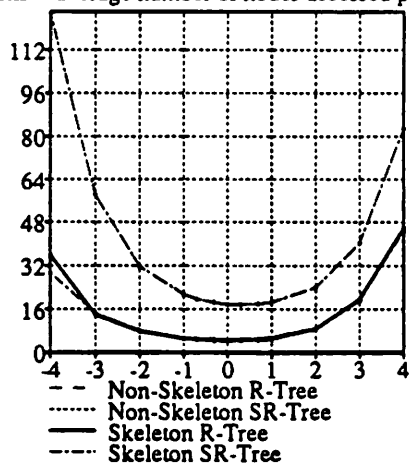
X Axis = horizontal/vertical query aspect ratio (log base 10)
Y Axis = average number of nodes accessed per search



Graph 3.9: Rectangle data;
Interval length distribution: Uniform [0, 100]

SEARCH PERFORMANCE OF 4 INDEX TYPES FOR 100,000
EXPONENTIALLY DISTRIBUTED (IN VERTICAL DIMENSION)
TUPLES, AS A FUNCTION OF THE QUERY ASPECT RATIO

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



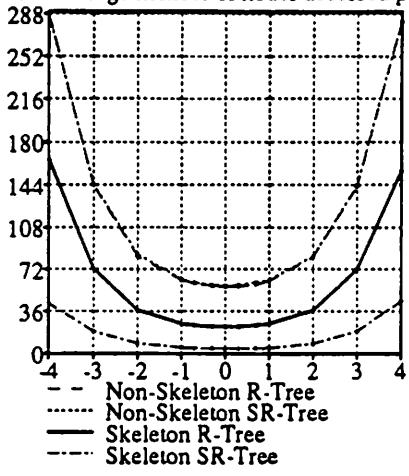
Graph 3.10: Rectangle data;
Interval length distribution: Uniform [0, 100]

Graph 3.11 shows the results of the four index types on the rectangle data distribution R3 that featured uniformly distributed interval center-points and 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. Another interesting point about these results is that the Skeleton SR-Tree outperformed the Skeleton R-Tree, but among the non-Skeleton Indexes there was no significant difference in performance between the SR-Tree and R-Tree. This was due to the small number of spanning rectangles that were present in the non-Skeleton SR-Tree, since the non-Skeleton SR-Tree had non-leaf nodes with large regions which reduced the likelihood of large rectangles spanning lower level nodes.

Graph 3.12 contains the results of the experiments involving distribution R4 of 10^5 tuples.

SEARCH PERFORMANCE OF 4 INDEX TYPES FOR
UNIFORMLY DISTRIBUTED DATA (100,000 TUPLES)
AS A FUNCTION OF THE QUERY ASPECT RATIO

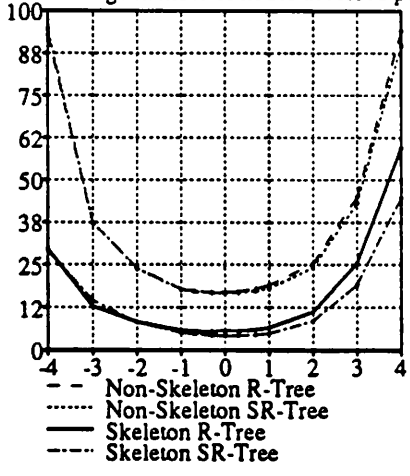
X Axis = horizontal/vertical query aspect ratio (log base 10)
Y Axis = average number of nodes accessed per search



Graph 3.11: Rectangle data;
Interval length distribution: Exponential(2000)

SEARCH PERFORMANCE OF 4 INDEX TYPES FOR 100,000
EXPONENTIALLY DISTRIBUTED (IN VERTICAL DIMENSION)
TUPLES, AS A FUNCTION OF THE QUERY ASPECT RATIO

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



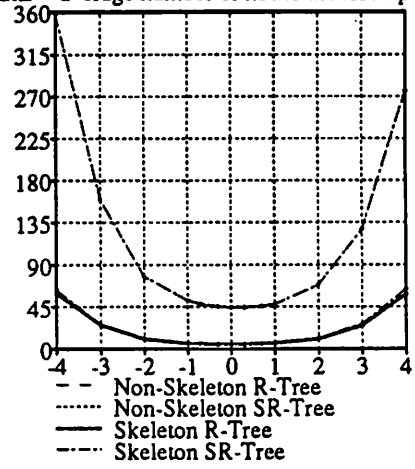
Graph 3.12: Rectangle data;
Interval length distribution: Exponential(200)

In this distribution, the interval center-points were exponentially distributed in the vertical dimension. In this graph, the Skeleton Indexes out-performed the non-Skeleton Indexes. This graph is similar to Graph 3.10 which featured data with interval lengths distributed over [0, 100], largely because the exponential interval length distribution had a parameter $\beta = 200$, as opposed to a value of 2000 used in the earlier experiments involving exponential interval lengths. The reason for the smaller value of β in this experiment was because Segment Indexes are inherently not well-suited to highly overlapping rectangle data such as a set of rectangles with an exponential interval *center-point* distribution as well as a highly exponential interval *length* distribution. This highly overlapping rectangle data contains a greatly concentrated number of spanning rectangles in one (the lower) end of the region. As previously discussed, the Segment Index implementations used a static partitioning of the fraction of branch entries on a node. In particular, 2/3 were dedicated to branch entries, and 1/3 for spanning index records. A static partitioning is a reasonable approach, since it ensures a minimum branching factor, and it eliminates the performance overhead and complexity of changing the partitioning dynamically. Whether the partitioning is static or dynamically controlled, it is necessary to maintain a certain minimum fanout (branching factor) per node to provide adequate search performance. Therefore, if the ratio of spanning index records to branch records greatly exceeds that of the original partitioning ratio, the performance of the index is negatively impacted due to a large increase in the number of non-leaf nodes that are necessary to accommodate the spanning index records. In particular, these "extra" nodes will have a large number of spanning index records and a small number of branch index records. In general, the Segment Indexes performed well when indexing highly overlapping rectangle data sets provided that the interval length distributions were not excessively non-uniform. This problem is compounded

as the dimensionality of the data increases. In the case of hyper-rectangular data (intervals in $K > 2$ dimensions) with highly non-uniform center-point and interval length distributions, there is a greater potential for the spanning index record to branch index record ratio to be large, since a data item may be stored in a spanning index record if it spans a child node in *any* dimension.

SEARCH PERFORMANCE OF 4 INDEX TYPES FOR
UNIFORMLY DISTRIBUTED DATA (200,000 TUPLES)
AS A FUNCTION OF THE QUERY ASPECT RATIO

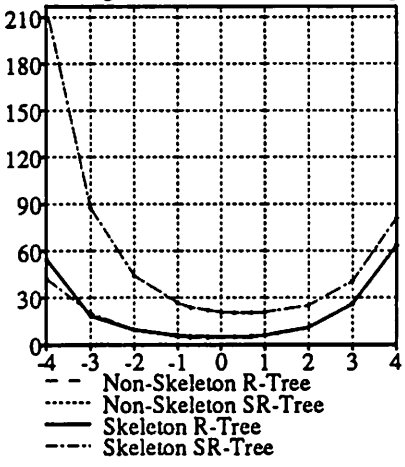
X Axis = horizontal/vertical query aspect ratio (log base 10)
Y Axis = average number of nodes accessed per search



Graph 3.13: Rectangle data;
Interval length distribution: Uniform [0, 100]

SEARCH PERFORMANCE OF 4 INDEX TYPES FOR 200,000
EXPONENTIALLY DISTRIBUTED (IN VERTICAL DIMENSION)
TUPLES, AS A FUNCTION OF THE QUERY ASPECT RATIO

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



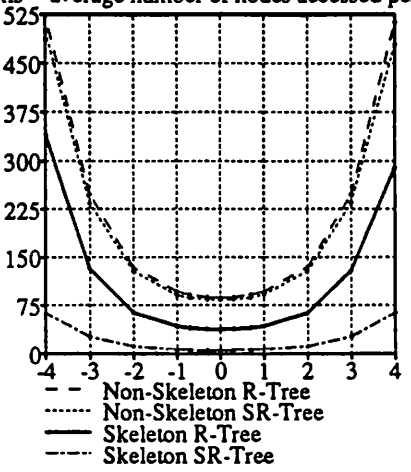
Graph 3.14: Rectangle data;
Interval length distribution: Uniform [0, 100]

Although Segment Indexes have difficulty dealing with rectangular data which is both highly non-uniform in its spatial location and interval lengths, such data collections pose problems for all spatial indexing structures. Any collection of data that is both highly overlapping and highly variable in length is difficult to index efficiently. R-Trees would suffer from massive overlap, and R+-Trees would become prohibitively large due to partitioning large rectangles into many small components. The idea of an index is to efficiently partition data that is itself readily partitionable. When the data to be indexed lacks partitionability, it is difficult if not impossible to devise a suitably efficient indexing structure for the data.

Graphs 3.13-3.16 correspond to the results of Graphs 3.9-3.12, except that the data sets contained 2×10^5 tuples for each of the data distributions R1-R4, rather than 10^5 tuples. The results of the experiments involving data sets of size 2×10^5 for each of the distributions R1-R4 were qualitatively similar to those in Graphs 3.9-3.12, and differed only in that the magnitude of the results were larger.

SEARCH PERFORMANCE OF 4 INDEX TYPES FOR
UNIFORMLY DISTRIBUTED DATA (200,000 TUPLES)
AS A FUNCTION OF THE QUERY ASPECT RATIO

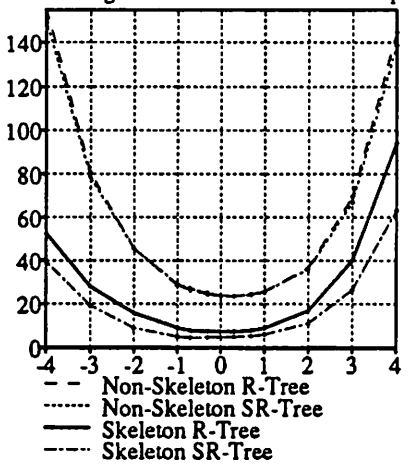
X Axis = horizontal/vertical query aspect ratio (log base 10)
Y Axis = average number of nodes accessed per search



Graph 3.15: Rectangle data;
Interval length distribution: Exponential(2000)

SEARCH PERFORMANCE OF 4 INDEX TYPES FOR 200,000
EXPONENTIALLY DISTRIBUTED (IN VERTICAL DIMENSION)
TUPLES, AS A FUNCTION OF THE QUERY ASPECT RATIO

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



Graph 3.16: Rectangle data;
Interval length distribution: Exponential(200)

3.6. 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, tree-structured indexes has been described. Aspects of the Segment Tree memory resident data structure were combined with the R-Tree database indexing structure to index large collections of multi-dimensional interval data. This index was referred to as the Segment R-Tree, or SR-Tree. The SR-Tree was further enhanced by a pre-allocation and pre-partitioning scheme which was referred to as a *Skeleton* SR-Tree. The Skeleton SR-Tree refinement to the basic SR-Tree is to pre-construct an index that provides an initial decomposition of the space to be indexed based on a given or estimated data distribution. The Skeleton SR-Tree adapts to the actual data distribution using splitting to handle node overflow and adjacent node coalescing for node underflow and thus the index is completely dynamic, i.e., it may adapt to any data distribution. Performance results comparing R-Trees, SR-Trees, Skeleton R-Trees, and Skeleton SR-Trees were presented which demonstrate that the Skeleton SR-Trees provide a substantial performance improvement over conventional indexing techniques for both rectangle and line segment data. These results suggest that for spatial access methods which are based on the *overlapping cells* technique, such as the R-Tree, the Segment Index variation of that access method would perform best when using the Skeleton Segment Index variant.

Skeleton Segment Indexes were shown to work well in most cases, i.e., when the data consists of line segment data or spatial data that is not simultaneously highly non-uniform in spatial location and interval length. The excluded case is not likely to be a severe limitation, since rectangular data that is highly non-uniform in both spatial location and interval length exhibit a high degree of overlap, and therefore are inherently difficult for any spatial data structure to index efficiently.

Furthermore, in most applications such data distributions are unlikely to occur in practice.

CHAPTER 4

LOP-SIDED INDEXES

4.1. Introduction

Indexing techniques for database management systems that are based on multi-way search trees are almost invariably balanced tree structures. A good example of such an index is the B-Tree [BAYE72] or one of its variants, such as the B+-Tree [KNUT73, COME79]. The design of tree-structured indexes based on balanced trees resulted from an assumption that the frequency of access is at least approximately equal for all tuples in the indexed relation. While this assumption of a *uniform query distribution* may be appropriate for many applications, there may be cases in which information on the probabilities of access for a particular key is known or may be estimated with high probability. If the entire set of data to be indexed were made available at once, the problem of constructing an optimal, multi-way search tree index would be straightforward. However, for most applications database indexing structures must be dynamic, allowing insertions concurrently with search operations. The subject of dynamic, *lop-sided* (unbalanced), multi-way, tree-structured indexes is subject of this chapter.

The concept of optimal multi-way search trees, or what will hereafter be referred to as *lop-sided trees* has been much studied for the case of a small node branching factor b, particularly for the case of $b = 2$, e.g., the Optimal Search Tree [KNUT73]. The subject of unbalanced trees has generally not been considered for values of b

significantly larger than 2, as would be typical of a multi-way tree with a high degree of node fanout. Such trees are typical of the data structures which are useful for database system indexes that are paged onto secondary storage. Current database systems have universally adopted balanced multi-way tree-structured indexes, such as the B-Tree, not only because of the aforementioned uniform access query distribution assumption, but also since because the number of levels of a "typical" database indexing structure is in the range of 3-5. For example, a database containing 10 gigabytes of data may be indexed by a 4-level B+-Tree with a node (page) size of 8 Kb. In order for a lop-sided index to perform better than a balanced index, it would have to contain a substantial fraction of the most recently accessed data in a sub-tree whose number of levels was less than that of a balanced index that contained the entire set of data.

Since the capacity of a balanced tree index grows exponentially in the number of levels, i.e., at a rate proportional to b^h where b is the node branching factor and h is the height of the index in levels, balanced tree indexes are generally useful because the number of levels grows as the $\log_b(n)$, where n is the number of tuples indexed. For balanced tree indexes that are built using large node sizes (such as 8 Kb), b may be in the range of 250-500, for an index record size between 16-32 bytes, respectively. Therefore, the height of a balanced index with a large node size grows rather slowly, as illustrated in Table 4.1 for various node (page) sizes, an index record size of 16 bytes, a node header of 12 bytes, and node utilizations of 69% and 100%. The capacity of an index with 69% utilization is of interest because empirical evidence resulting from the performance experiments reported later in this chapter involving B+-Trees showed that the average node utilization was 69% when insertions (but no deletions) were performed. This result closely corresponds with the approximate estimate of 66% given in [BAYE72].

Page Size (Kb)	Index Height (levels)	Index Capacity (69% util)	Index Capacity (100% util)
1	1	43	63
1	2	1849	3969
1	3	79507	250047
1	4	3.4188e+06	1.5753e+07
1	5	1.47008e+08	9.92437e+08
1	6	6.32136e+09	6.25235e+10
1	7	2.71819e+11	3.93898e+12
2	1	88	127
2	2	7744	16129
2	3	681472	2.04838e+06
2	4	5.99695e+07	2.60145e+08
2	5	5.27732e+09	3.30384e+10
2	6	4.64404e+11	4.19587e+12
2	7	4.08676e+13	5.32876e+14
4	1	176	255
4	2	30976	65025
4	3	5.45178e+06	1.65814e+07
4	4	9.59513e+08	4.22825e+09
4	5	1.68874e+11	1.0782e+12
4	6	2.97219e+13	2.74942e+14
4	7	5.23105e+15	7.01102e+16
8	1	354	511
8	2	125316	261121
8	3	4.43619e+07	1.33433e+08
8	4	1.57041e+10	6.81842e+10
8	5	5.55925e+12	3.48421e+13
8	6	1.96797e+15	1.78043e+16
8	7	6.96663e+17	9.09801e+18

Table 4.1: Index Capacities for Various Node Sizes

With the recent advent of optical disk storage technology, it has become cost-effective to maintain large historical data archives on-line. Large historical data archives have precisely the characteristics that make lop-sided indexes advantageous over balanced indexes. In particular, the amount of data will tend to be very large, and the distribution of queries over such data is likely to be highly non-uniform, e.g., queries on more recent historical data are likely to occur more frequently than

queries on older historical data.

The remainder of this chapter proceeds as follows. Section 2 describes the structure of a lop-sided index. Section 3 explains the limitations of a large branching factor in optimal unbalanced binary search tree techniques. Section 4 defines the concepts of *upward* and *downward* node splitting, which are the mechanisms for dynamically constructing a lop-sided index. Section 5 and 6 present the structure and algorithms for a lop-sided B+-Tree index, respectively. Section 7 describes a performance study that was carried out which compared the performance of a lop-sided B+-Tree to a balanced B+-Tree for a range of non-uniform query distributions, and presents the results from that study. Section 8 discusses generalizations and future research issues related with the lop-sided index concept. Section 9 contains a summary and conclusions.

4.2. Structure of a Lop-Sided Index

A balanced tree has some number of levels from the root to a leaf node. One approach to building a lop-sided tree is to combine a number of balanced sub-trees in a configuration that constitutes a lop-sided tree, as illustrated in Figure 4.1. This approach was followed in this research, since the overriding constraint is that a lop-sided index must perform as well or better than a balanced index on at least some non-uniform query distributions. Given this constraint, the height of the most frequently accessed sub-tree(s) must be less than the height of a corresponding balanced index, and their capacities must be large enough to contain some subset of the most frequently accessed records. These two requirements are best served simultaneously by constructing a lop-sided tree consisting of some number of balanced sub-trees of height $h-1$ which contain the most frequently accessed records, where h is the height of the corresponding balanced index.

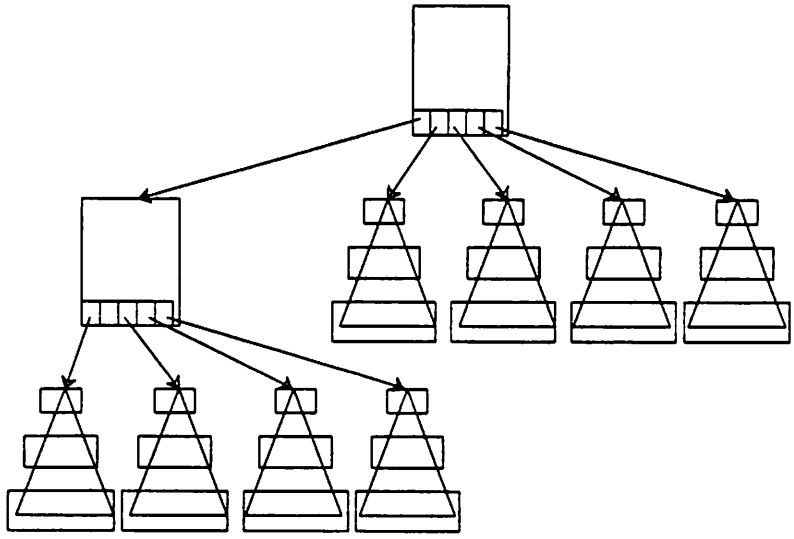


Figure 4.1: Lop-sided Index

4.3. Limitations of Large Branching Factor in Unbalanced Binary Search Trees

There have been several approaches for optimal weighted binary search trees, such as the Optimal Binary Search Tree [KNUT73], BB(α)-Tree [NIEV73], Splay Tree [SLEA85], and the Randomized Search Tree [ARAG89]. The difficulty with extending such schemes to multi-way, *b*-ary trees for $b > 2$ is that a substantial amount of tree reorganization would be required whenever a node was moved from a lower to a higher level of the tree, or vice versa. The problem of unbalanced *b*-ary search trees for large *b* is a new challenge, which has been overlooked until now since balanced multi-way indexes have adequately served database systems until recently. With the advent of massively larger on-line storage archives, however, balanced tree indexes are no longer the ubiquitous solution for nearly all collections of data, as they once were considered to be [COME79].

4.4. Node Splitting Dynamics: Splitting Down Versus Splitting Up

In balanced, multi-way, tree-structured indexes such as B-Trees, the data are stored in the leaf nodes, and the index grows in a bottom-up fashion using the conventional technique of *upward splitting*. Upward splitting is illustrated in Figure 4.2, and works as follows. When a node N is full and an attempt is made to store a new item on that node, the node is *split* and replaced by a new instance of itself N' and a new sibling node N'', and the original contents of node N plus the new item to be inserted which caused N to overflow are evenly distributed between N' and N''. Finally, a new branch pointer for N'' is installed in the parent node P of node N. If there is no room in P for the branch pointer for N'', node P is also split and the splitting of N is said to have propagated upward. In general, splits may propagate up to the root node, in which case the height (number of levels) of the index increases by

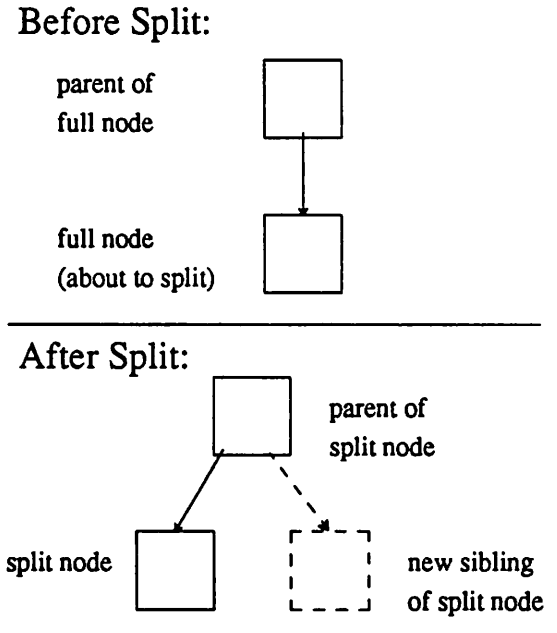


Figure 4.2: An upward split

one.

Upward splitting has many virtues, most notably that it maintains a balanced tree structure, it automatically partitions the higher-level nodes of the index according to the input data distribution, and it results in an index that is approximately 66% utilized on the average when no deletions are performed [BAYE72].

The alternative to upward splitting is *downward splitting*, which is illustrated in Figure 4.3. In a downward split, the branch pointer in the parent node *P* of the split node *N* is replaced by a pointer to a new non-leaf node *NP*, which acts as the new parent of *N*. Then, the sibling pair *N* and *N'* resulting from the split of *N* are installed as the children of node *NP*.

When upward splitting does not propagate up to the root node, it does not influence the number of levels in the tree. When upward splitting does propagate up to the root node, it adds one level to all search paths from the root to a leaf node. On the other hand, a downward split only adds one level along some subset of search paths from the root to a leaf node.

Downward splitting does not share the virtues of upward splitting, i.e., it does not provide any of the following benefits: balanced tree maintenance, automatic higher-level node partitioning, or minimum node utilization maintenance. It is due to these shortcomings that downward splitting is generally not applied to tree-structured database indexes. Downward splitting does have one important redeeming feature, however, namely that it provides a mechanism for the dynamic growth of a lop-sided index.

If both upward and downward splitting are to be used in the evolution of an index, the question arises as to how to determine whether a given node split should be performed upward or downward in order to produce a lop-sided index with a

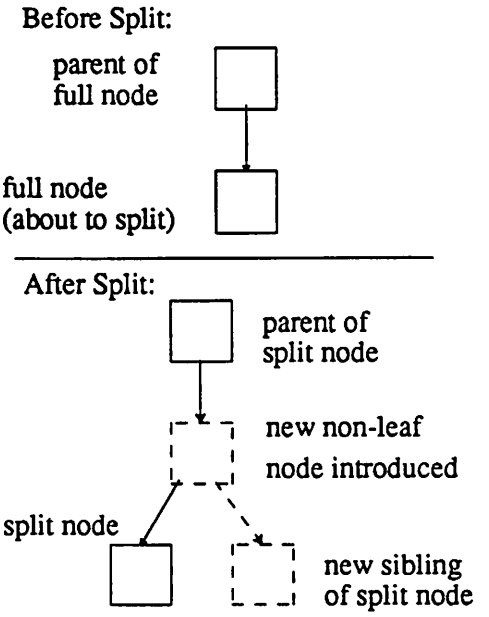


Figure 4.3: A downward split

specified *desired degree of lop-sidedness*. If the decision to split a node up or down is to be made dynamically based on local information (i.e., on information contained only in the node *N* to be split, *N*'s parent, and possibly a small number of *N*'s siblings or children), the index may occasionally require reorganization to maintain the desired degree of lop-sidedness. Even if global information is used, the correct decision of whether to split up or down cannot be made with certainty without knowing the future input data distribution. Therefore, some form of lop-sided tree reorganization is required in order to maintain a properly lop-sided index, i.e., one that maintains the desired degree of lop-sidedness as the size of the index changes.

In the following sections, algorithms for three variations of a lop-sided B+-Tree are described. The first variation does no tree reorganization, the second pre-allocates an initial indexing structure to reduce the potential necessity of

reorganization, and the third performs a limited type of reorganization when necessary. These variations will be referred to as the *no-shuffle*, *skeleton*, and *shuffle* methods. All three variations share a common data structure, which is presented in the next section.

4.5. Structure of a Lop-Sided B+-Tree Index

The structure of a lop-sided index version of a B+-Tree is presented in this section. As in the original B+-Tree, there are two types of nodes: non-leaf nodes and leaf nodes. These node types are illustrated in Figure 4.4.

The non-leaf nodes contain disk address pointers to other nodes, either to non-leaf nodes or to leaf nodes, and each such pointer is part of an index record which also contains a key value range. In addition, each non-leaf node *N* in the lop-sided B+-Tree contains *auxiliary information* consisting of *N*'s depth (in levels) from the root node, and *N*'s *partition number*. Assuming the probability distribution function of

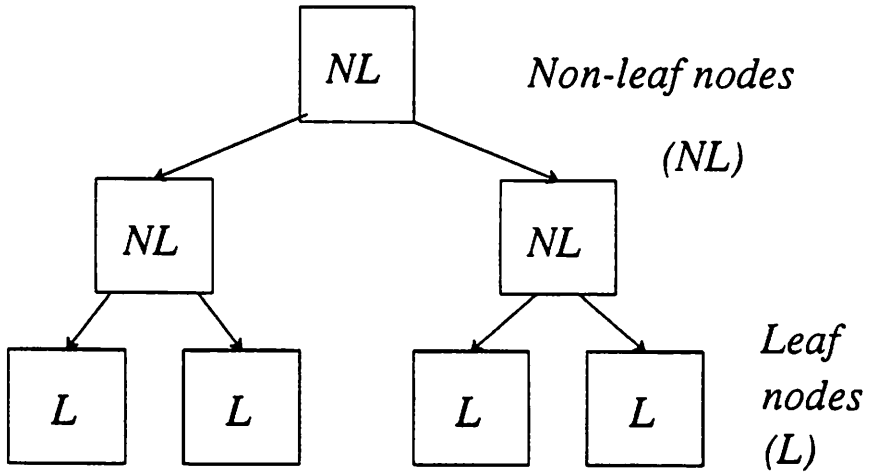


Figure 4.4: Leaf and Non-leaf Nodes

query values is known, *partitions* are used to divide the indexed domain into equal sized intervals so that the information contained in a specified query probability distribution function (QPDF) may be used to distinguish between nodes which are expected to be frequently accessed from those which are not. The number of partitions is controlled by a parameter, *NPARTS*, which determines the number of ranges of query values that may be assigned distinct probabilities according to the QPDF, and the partition size is determined by dividing the difference between the maximum and minimum values of the indexed domain by *NPARTS*. The partition number of a non-leaf node *N* refers to the partition that the interval corresponding to *N* maximally intersects.

Leaf nodes are always at the lowest level of a path that descends from the root, and each stores a set of index records which contain a key value and a tuple identifier (a tuple ID, or TID, is a disk pointer to a page containing the associated tuple). The fundamental difference between the structure of a lop-sided B+-Tree and a normal (balanced) B+-Tree is that the latter is a balanced tree, whereas the former is not constrained to be balanced.

One additional piece of bookkeeping information is required in order to maintain a lop-sided index, namely, a table containing the capacities of a balanced index with branching factor equal to that of the lop-sided index at each of various levels and fill factors, as exemplified by Table 4.1. This information is used to estimate the height that a balanced index would have if one were used to index the same amount of data contained in the lop-sided index. Hereafter, this table will be accessed by a function *height_bal*(*Ntuples*), which given a number of tuples (*Ntuples*), returns the height of a balanced index that could accommodate *Ntuples* records.

4.6. Algorithms for a Lop-Sided B+-Tree Index

This section describes the algorithms for node insertion, splitting, search, and deletion for the lop-sided B+-Tree whose structure was presented in the preceding section, for each of the methods of tree reorganization: no-shuffle, skeleton, and shuffle.

4.6.1. No-Shuffle Method

The *no-shuffle* lop-sided index is the simplest form of the lop-sided index, since it does no reorganization if the lop-sidedness of the index with respect to a balanced index cannot be maintained in its present configuration. This method was explored to understand its limitations in order to refine the indexing algorithms to cope with the reorganization problem.

The fundamental goal of this index is to use the specified query distribution to distinguish between *high-frequency access* (HFA) nodes and *low-frequency access* (LFA) nodes. HFA nodes have a higher than average probability of access, and LFA nodes have a lower than average probability of access. HFA nodes are kept within *height_bal*(*Ntuples*) – 1 levels (distance from the root node in number of levels) as long as possible.

While upward and downward splits are used in the same index, once an upward split is undertaken and propagates up to a higher level, if the higher level node splits then that node is also split upward. This policy decision was made in order to avoid the complexity of propagating upward splits that may trigger downward splits, or vice versa. Also, before an upward split is undertaken, a check is made to determine if the upward split would cause a propagation up to the root. This check is easily made during the depth-first traversal of the index to the node targeted for insertion that will initiate the splitting process. The purpose of this check will be made

apparent in the discussion of the node splitting algorithm.

4.6.1.1. Insertion Algorithm

The index begins with a single leaf node. When the first leaf node overflows, it is split upward, and the new root node is a non-leaf node. The index continues to grow by upward splits until the number of non-leaf node record entries becomes greater than or equal to *NPARTS*. At that time, the *auxiliary information* in the index is initialized, and is thereafter maintained by the splitting algorithm, which may decide to split up or down. For each non-leaf node *N*, the auxiliary information contains *N*'s depth from the root, and *N*'s partition number which specifies which portion of the indexed domain *N* maximally intersects. All insertions are performed following the conventional B+-Tree algorithm, i.e., choosing at each non-leaf node the branch in the tree whose range contains the value being inserted. When a leaf node is reached, an index record is inserted, and the count of records on that node is incremented by one. If the leaf node is full, it is *split*. The split algorithm is described in the next section.

4.6.1.2. Node Splitting Algorithm

The node splitting algorithm of the lop-sided B+-Tree controls the degree of lop-sidedness of any particular part of the index. When a node *N* splits, the auxiliary information contained in node *N* is used to decide whether *N* should be split up or down. This decision is made as follows. The first piece of information that is used in this decision is the *depth*(*N*), i.e. the depth of node *N*, which is contained in the auxiliary information part of the node. The second determining factor in the up-or-down splitting decision is whether *N* is in a HFA node or a LFA node. Assuming node *N* is in partition *i*, and that the QPDF is specified by a given probability distribution function *F*, *N* is defined to be a HFA node if:

$$Pr(a_i < Q \leq b_i) = F(b_i) - F(a_i) > 1 / NPARTS$$

where a_i and b_i are the lower and upper limits of partition number i, respectively. Otherwise, N is a low-frequency access node. In other words, a node is an HFA node if its query access probability is higher than the average for all partitions, and is a LFA node otherwise.

The third determining factor is whether or not an upward split would propagate up to the root, which is calculated during the depth-first traversal of the index. This determination is made by checking each node accessed along the path from the root to leaf node N to determine if all nodes along the path were full. If they all were full, then the *split_up_to_root* flag is set to TRUE, and FALSE otherwise.

The information consisting of *depth*(N), the classification of N as either an HFA or LFA node, and the setting of the *split_up_to_root* flag are then used to decide whether N should be split up or down. The decision is based on a goal of keeping HFA nodes within *height_bal*(*Ntuples*) – 1 levels for as long as possible. The decision is made as follows (in pseudo-C notation):

```
if (N is an HFA node) then
        if (split_up_to_root is FALSE) then
                split up
        else  /* split_up_to_root is TRUE */
                if (depth(N) < height_bal(Ntuples) - 1) then
                        split up
                else
                        split down
else  /* N is LFA node */
        split down
```

If N is an HFA node, it is split up, unless an upward split would propagate to the root and N's depth is at least one less than the height of a balanced tree containing the same number of records, in which case it is split down. The reason for splitting an HFA node down in this case is to allow the leaf nodes that are still at a depth less than *height_bal*(*Ntuples*) to remain shallower than the height of a balanced tree for

as long as possible. The reason for splitting an HFA node *N* up when *split_up_to_root* is TRUE and *depth*(*N*) < *height_bal*(*Ntuples*) – 1 is that splitting up in this case will maintain *N* within *height_bal*(*Ntuples*) – 1 levels from the root even though the index will grow in height by one level as the root must be split up.

The general strategy is that HFA nodes are split up and LFA nodes are split down, except for the special cases mentioned above. The reasoning behind this strategy is that upward splitting does not cause the depth of any leaf to change unless the root splits, in which case the depth of all leaf nodes increases by one. Downward splitting, on the other hand, only increases the depth of a subset of leaf nodes. Therefore, downward splitting is most applicable for LFA nodes, and upward splitting is suited for HFA nodes.

4.6.1.3. Search Algorithm

The lop-sided B+-Tree search algorithm is identical to that of the original B+-Tree search algorithm, with the exception that the number of node accesses per search is not equal in all cases, but varies with the depth of the leaf node accessed. If a balanced B+-Tree index containing the same number of tuples as a lop-sided B+-Tree has height h, in the lop-sided index the HFA nodes would generally be at a depth of $h-1$ or h, whereas LFA nodes would generally be at a depth of h of $h+1$.

4.6.1.4. Deletion and Underflow Algorithms

To delete a record from a lop-sided B+-Tree, the record is found using the search algorithm, and then the record is marked as being empty, and the record count on the node is decremented by one. The deletion algorithm is similar to the insertion algorithm, except that rather than dealing with the possibility of node *overflow* which is handled by splitting the node, when deleting a record the possibil-

ity of node *underflow* arises. Node underflow is defined as a node whose record count has gone below a certain threshold minimum utilization parameter. Typical minimum utilization parameter settings are 1/2 or 1/3.

To handle node underflow, the inverse of the splitting algorithm is performed. Following a deletion, if the utilization of the node N that contained the deleted record goes below the minimum utilization parameter, the contents of node N are transferred into its adjacent sibling node S at the same depth as N which has the fewest entries, as long as this transfer would not result in an overflow of node S. If the node S would overflow if it were merged with N, rather than transferring the contents of N to S, the contents of both N and S are evenly distributed between the two nodes.

The underflow algorithm for node underflow stated above is not yet complete, as underflow may propagate up the tree. After the above algorithm is applied, if N was merged into a sibling node S, if S's parent node P underflows as a result, the underflow algorithm is applied to P and its least utilized adjacent sibling. This process may propagate all the way up to one level below the root node, whose minimum number of children may be as low as two. If the number of children in the root node becomes one, the root node is replaced by its (only) child.

4.6.2. Skeleton Method

The *skeleton* method of lop-sided indexes has similarities to the *Skeleton Indexes* of Chapter 3. The application of the skeleton approach in this context is that given an estimate on the amount of data to be indexed, the lop-sided index may be pre-constructed based on this estimate so that the maximum number of HFA nodes may be stored at a depth less than that of a balanced index. This is accomplished by pre-constructing an index so that there is a sufficient number of levels for

the LFA nodes to populate without filling the capacity of the HFA nodes. In addition, the resolution (range of values that divide the index key domain among the high-level nodes) of the intervals in each of the non-leaf nodes reflect the capacity of the sub-trees which together constitute the initial configuration of the pre-allocated nodes. The algorithms from the *no-shuffle* method are used without modification in the skeleton method lop-sided index. The only refinement is the pre-allocation of the index, which is performed as follows.

The estimated number of tuples, *Etuples*, is used to determine the height of a balanced index if one were used to index the data, by the function *height_bal*(*Etuples*). The specified query distribution is then analyzed by dividing the index key domain into *NPARTS*, and classifying the partitions into HFA and LFA partitions. The *Etuples* value is then divided by the expected node occupancy of 0.69 times the branching factor (b) to obtain *Enodes*, the estimated number of nodes in the index. Then, the estimated number of LFA nodes is calculated by:

$$Elfa = Enodes \times \frac{number\ of\ LFA\ partitions}{NPARTS}$$

and the estimated number of HFA nodes is:

$$Ehfa = Enodes \times [1 - \frac{number\ of\ LFA\ partitions}{NPARTS}]$$

After these calculations, a root node is pre-allocated, and then *Elfa* nodes are pre-allocated at a depth of up to *height_bal*(*Etuples*) + 1, in a configuration corresponding to the query distribution (based on the partition classifications). Similarly, *Ehfa* nodes are pre-allocated at a depth of up to *height_bal*(*Etuples*) – 1. Finally, the interval values in each of the descendant node branch pointers contained in the non-leaf nodes are filled in to reflect the distribution of data contained in the index. Based on the values of *Etuples*, *Elfa*, and *Ehfa*, and the configuration of the index (which was

based on the query distribution), the values of the intervals that correspond to the data contained in each node are initialized.

This strategy results in an index that has a greater capacity to accommodate HFA nodes that are at a shallower depth than the height of a balanced index as compared to a no-shuffle method lop-sided index. This is because lop-sided indexes lose their advantage over a balanced index when the capacity of the "one level closer" portions of the index that contain the HFA nodes are exceeded. By pre-allocating the index based on an estimate of the number of tuples and the query distribution, the interval ranges in the root node for the HFA nodes can be pre-determined such that only the amount of data that will fit within *height_bal*(*Etuples*) – 1 levels may be accommodated.

4.6.3. Shuffle Method

The shuffle method lop-sided index periodically performs a limited form of tree reorganization to maintain the lop-sidedness of the index. This is accomplished by *shuffling* a set of nodes to a greater depth in the tree to make room for more entries at the shallower depth. For example, an example of a node *N* being shuffled down one level is illustrated in Figure 4.5.

The algorithms for insertion and search for the shuffle method lop-sided B+-Tree index are the same as those for the no-shuffle method lop-sided B+-Tree. The splitting and underflow algorithms are modified, as described in the following sections.

4.6.3.1. Splitting Algorithm

The modification to the no-shuffle splitting algorithm for the shuffle method lop-sided index occurs in the second else clause below.

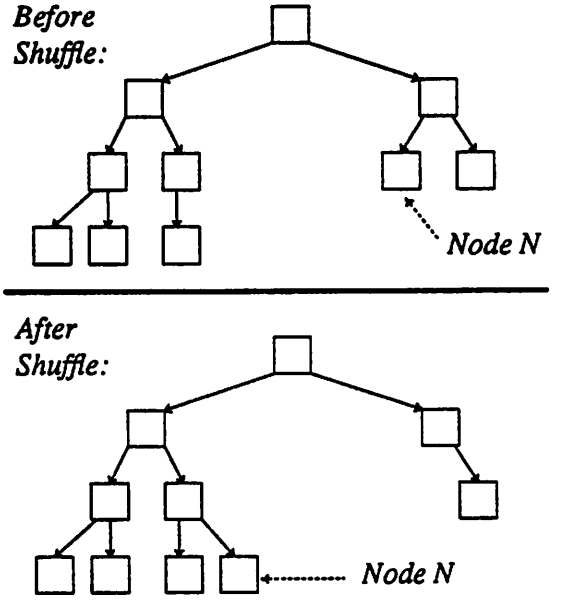


Figure 4.5: *Shuffling* a Node

```
if (N is a HFA node) then
        if (split_up_to_root is FALSE) then
                split up
        else  /* split_up_to_root is TRUE */
                if (depth(N) < height_bal(Ntuples) - 1) then
                        split up
                else
                        shuffle a child of root down to a depth 1 node
else  /* N is LFA node */
        split down
```

When the splitting node *N* is a HFA node, *split_up_to_root* is TRUE, and *depth*(*N*) ≥ *height_bal*(*Ntuples*) – 1, rather than splitting *N* down as was done in the no-shuffle method, a child of the root is shuffled down to a node at depth one. This may in turn cause a child of the depth one node to be shuffled down to a depth two node, etc. In the worst-case, *d* shuffles will propagate from the root to a leaf node at depth *d*, where the root is defined to be at depth zero.

4.6.3.2. Deletion and Underflow Algorithms

The deletion algorithm is unchanged from that of the no-shuffle lop-sided index. The underflow algorithm has one modification. Rather than restricting the potential candidates for merging to be the adjacent sibling nodes at the same depth as *N*, the candidates may also include adjacent sibling nodes that are one level higher or lower than the depth of *N*.

4.7. Performance Study

In order to test the ideas presented above, simulations of the lop-sided B+-Trees which employ the no-shuffle, skeleton, and shuffle methods were constructed, and their search performance characteristics were measured. A description of the performance experiments and the experimental results generated from them are presented in the following sections.

4.7.1. Simulation of Lop-Sided B+-Tree Indexes

Lop-sided indexes were simulated by implementing the algorithms stated above for insert, node splitting, and search. The input parameters were the node (page) size, the index record size, and the maximum threshold depth (MTD) within which the lop-sided index attempts to maintain HFA nodes. The input parameters used in each of the experiments are reported in the next section which presents the results of the performance experiments.

In all of the simulation experiments, the observed average node occupancy counts were approximately 69%[1] times *b*, where *b* is the branching factor of a node

---

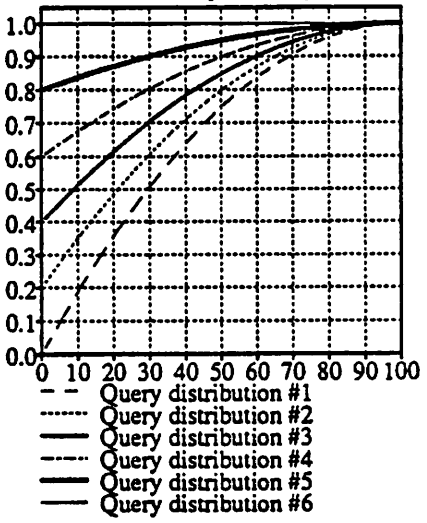
[1]This agrees with the estimate given in [BAYE72]: "In an index without deletions overflows will increase the storage utilization in the worst cases to about 66%." In the experiments reported here, the storage utilization was approximately 69%.

which is calculated by dividing the node size minus the size of the node header by the size of an index record. For comparison purposes, the number of levels of the corresponding balanced index was also computed.

The amount of indexed data was varied so that the size of each lop-sided index ranged from just over the capacity of a balanced index whose height was that of the maximum threshold depth (MTD) parameter, to that of the capacity of a balanced index whose height was equal to MTD + 1. The domain of the indexed data was uniformly distributed over $[0, 10^5]$. The number of partitions, *NPARTS*, used to divide the index key domain into HFA and LFA partitions was empirically chosen to be 100. This value for *NPARTS* provided good results, and the resulting partition size was therefore 1000. Recall that the purpose of partitioning is to divide the indexed domain into probability "buckets" for the purpose of transforming the continuous QPDF into a series of discrete partitions for distinguishing HFA/LFA nodes. The tradeoff of choosing *NPARTS* is that the partition size should be small enough to accurately reflect the continuous QPDF, and yet large enough so that the maximal intersection between the intervals corresponding to nodes and the set of partitions may be determined efficiently. In particular, it is desirable that each node intersect a small number of partitions.

QUERY DISTRIBUTION FUNCTIONS OF
QUERIES USED IN EXPERIMENTS

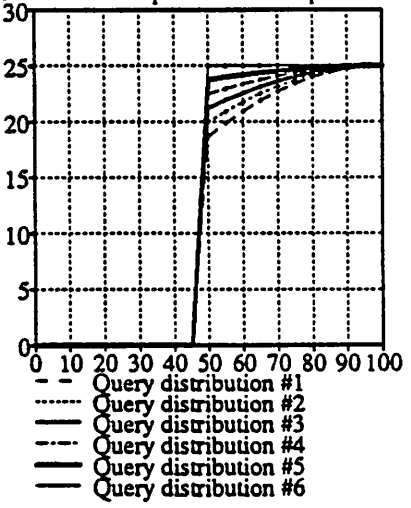
X Axis = percentage of indexed data
Y Axis = cumulative probability (Pr(Q <= q))



Graph 4.1: Query Distribution Functions

RELATIVE SEARCH PERFORMANCE IMPROVEMENT OF
LOP-SIDED INDEXES AS A FUNCTION OF THE UNUSED
CAPACITY OF L-TH LEVEL OF BALANCED INDEX

X Axis = unused capacity of L-th level of balanced index (percent)
Y Axis = relative search performance improvement (percent)



Graph 4.2: $L = 5$, Threshold level = 4, no-shuffle method.

The six query distributions which were used in the performance experiments are illustrated in Graph 4.1. Graph 4.1 contains the cumulative probability distribution functions for the six query distributions, i.e., for a random variable Q, the probability that $Q \leq q$, where q is a given point on the index key domain. The six query distribution ranged from the *arithmetic progression query distribution* (defined in Equation 4.1) to the *full-tilt* maximally lop-sided distribution that always searches for data contained in one partition of the index key domain. For example, if the most frequently accessed queries occur in the first partition, the probability distribution function for the full-tilt query distribution would be: {1, 0, 0, 0, ...}. The arithmetic progression query distribution was calculated by solving the following for x:

$$\sum_{t=1}^{npartitions} \left(\frac{t}{npartitions}\right) \times x = 1 \qquad 4.1$$

The four distributions between the arithmetic progression query distribution and the full-tilt distribution were generated by linear extrapolation between the arithmetic progression and full-tilt distributions.

Once the six query distributions were generated, for each query distribution, 100 random point queries were performed on the lop-sided index according to the specific query distribution, and the relative performance improvement (or degradation) provided by the lop-sided index with respect to the corresponding balanced index was calculated and recorded.

In the next section, the results of the performance experiments are presented.

4.7.2. Performance Results from Simulation

The lop-sided tree simulations were used to compare various lop-sided indexes to their balanced index counterparts.

Node Size (Kb)	Branch Size (bytes)	Maximum Threshold Depth	Branching Factor
1	16	4	43
4	16	3	176
8	16	2	354

Table 4.2: Simulation Parameters

In all of the experiments, it was assumed that the root node of either the balanced or lop-sided index was buffered in main memory, which effectively reduced the number of accesses per search for all the index types by one. In practice, it is likely that more than just the root nodes would be buffered. However, in the interest of making a fair comparison between the two index types, only the root node was assumed to be buffered.

In Graphs 4.2-4.10, the performance improvement (or degradation) provided by the lop-sided index relative to a corresponding balanced index is plotted as a function of the unused capacity of an L-level balanced index, where L was set to 5. The maximum threshold depth (MTD) parameter of the lop-sided index was set to 4. The graphs were plotted in this way because this showed the difference in performance between a lop-sided and a balanced index over an entire cycle, i.e., over the range of the amount of indexed data that would fill a $L-1$ level balanced index to a L level balanced index. When the balanced index would grow to $L+1$ levels, the cycle shown in the performance graphs would repeat, albeit with different magnitudes. A succession of three such cycles are presented in Graphs 4.2-4.4, 4.5-4.7, and 4.8-4.10, respectively.

In this and all of the graphs discussed in this section, the six query distributions are labeled 1 through 6, where query distribution 1 corresponds to the

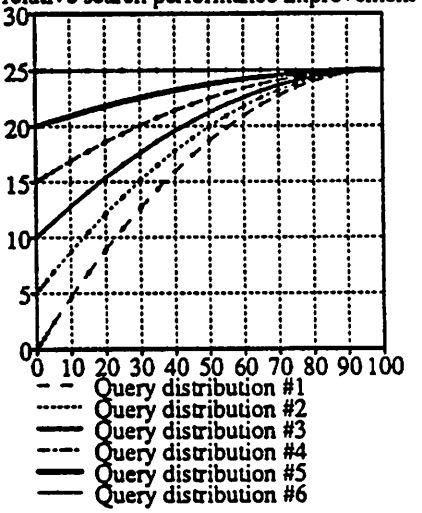
arithmetic progression query distribution, and query distribution 6 corresponds to the full-tilt query distribution.

The input parameters to the simulations which were used in the experiments are given below in Table 4.2.

Graphs 4.2-4.4 show the performance results for lop-sided indexes with a maximum threshold number of levels of 4 for the no-shuffle, skeleton, and shuffle methods, respectively. In Graph 4.2, the no-shuffle lop-sided index provided an improvement over the balanced index when the unused capacity of the fifth level of the balanced index was greater than 50%. When the number of tuples exceeded that amount, the capacity of the most frequently accessed nodes that were maintained within a depth of four levels became exceeded, and the HFA nodes were split down. At that point, the root node was split down because the HFA nodes exceeded their capacity at a depth of four, and the index essentially was transformed into a five-level balanced index.

RELATIVE SEARCH PERFORMANCE IMPROVEMENT OF LOP-SIDED INDEXES AS A FUNCTION OF THE UNUSED CAPACITY OF L-TH LEVEL OF BALANCED INDEX

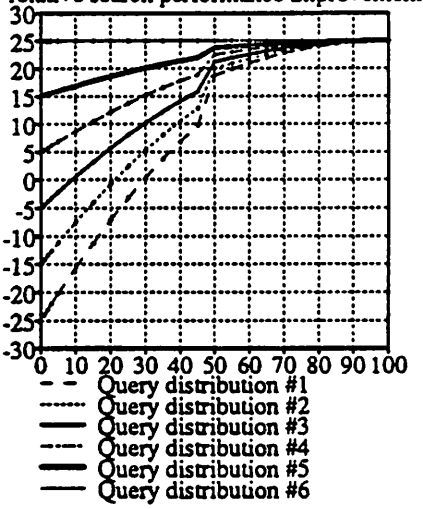
X Axis = unused capacity of L-th level of balanced index (percent)
Y Axis = relative search performance improvement (percent)



Graph 4.3: L = 5, Threshold level = 4, skeleton method.

RELATIVE SEARCH PERFORMANCE IMPROVEMENT OF LOP-SIDED INDEXES AS A FUNCTION OF THE UNUSED CAPACITY OF L-TH LEVEL OF BALANCED INDEX

X Axis = unused capacity of L-th level of balanced index (percent)
Y Axis = relative search performance improvement (percent)



Graph 4.4: L = 5, Threshold level = 4, shuffle method.

However, while the unused capacity of the fifth level of the balanced index was greater than 50%, the relative performance improvement provided by the lop-sided index was as much as 25%, since the height of the balanced index was five and some fraction of HFA nodes were maintained within a depth of four, and the root node access was not counted in either case due to the assumption that it is buffered in main memory.

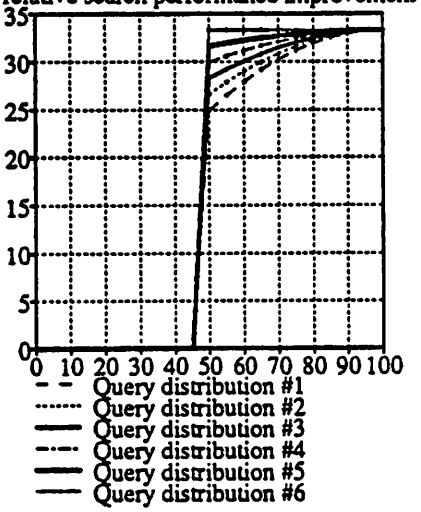
Graph 4.3 shows the results of the skeleton method experiments. In these experiments, the initial pre-constructed index configuration was based on an estimated number of tuples equal to the capacity of a five-level balanced index. The performance improvement relative to a balanced index that was provided by the skeleton lop-sided index were proportional to the query distributions in Graph 4.1, ranging between 0 and 25%.

Graph 4.4 shows the results of the shuffle method experiments. In Graph 4.4, the results are the same as in Graphs 4.2 and 4.3 when the unused capacity of the balanced index was at least 50%. Below that point, the performance improvement gradually diminished due to a downward split in a child of the root node that caused the LFA part of the index to grow down by one level. Although three of the query distributions experienced a negative performance impact over part of the balanced index capacity range, the three most lop-sided query distributions experienced a performance improvement over the entire range.

Graphs 4.5-4.7 correspond to Graphs 4.2-4.4, except that the height of the balanced index is four and the maximum threshold depth of the lop-sided index is three.

RELATIVE SEARCH PERFORMANCE IMPROVEMENT OF LOP-SIDED INDEXES AS A FUNCTION OF THE UNUSED CAPACITY OF L-TH LEVEL OF BALANCED INDEX

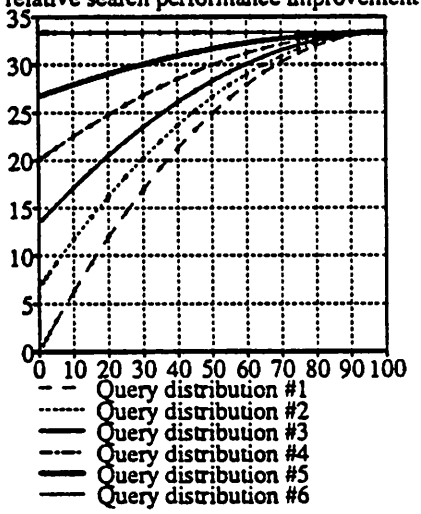
X Axis = unused capacity of L-th level of balanced index (percent)
Y Axis = relative search performance improvement (percent)



Graph 4.5: L = 4, Threshold level = 3, no-shuffle method.

RELATIVE SEARCH PERFORMANCE IMPROVEMENT OF LOP-SIDED INDEXES AS A FUNCTION OF THE UNUSED CAPACITY OF L-TH LEVEL OF BALANCED INDEX

X Axis = unused capacity of L-th level of balanced index (percent)
Y Axis = relative search performance improvement (percent)



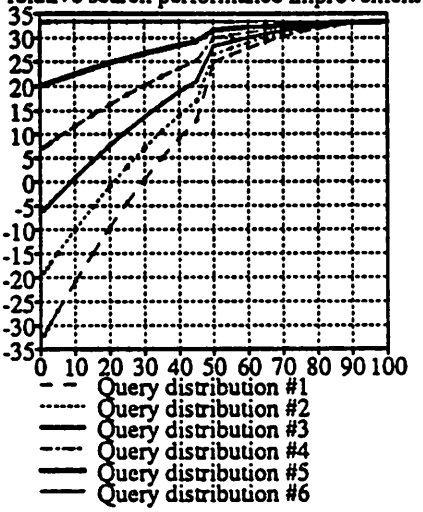
Graph 4.6: L = 4, Threshold level = 3, skeleton method.

Similarly, Graphs 4.8-4.10 correspond to Graphs 4.2-4.4, except the balanced index height is three and the lop-sided index maximum threshold depth is two. The results in these sets of graphs are qualitatively similar to Graphs 4.2-4.4, except the maximum relative performance improvement is 33% in Graphs 4.5-4.7 and 50% in Graph 4.8-4.10, reflecting the height of the indexes in the respective cases.

Comparing the results of Graphs 4.2-4.10, the skeleton method provided the best overall results, followed by the shuffle method, and then the no-shuffle method. These results are to be expected, since given a reasonably accurate (i.e., to within an order of magnitude) estimate of the number of tuples to be indexed, the skeleton method can always outperform either of the other methods, since both the no-shuffle and shuffle methods suffered from downward split expansion which resulted in lop-sided indexes that were in part equal to or beyond the depth (L) of a balanced index when the percentage of unused capacity of the L-th level of the balanced index reached 50%.

RELATIVE SEARCH PERFORMANCE IMPROVEMENT OF LOP-SIDED INDEXES AS A FUNCTION OF THE UNUSED CAPACITY OF L-TH LEVEL OF BALANCED INDEX

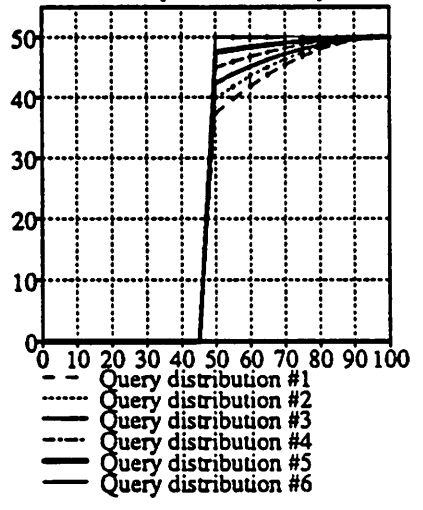
X Axis = unused capacity of L-th level of balanced index (percent)
Y Axis = relative search performance improvement (percent)



Graph 4.7: L = 4, Threshold level = 3, shuffle method.

RELATIVE SEARCH PERFORMANCE IMPROVEMENT OF LOP-SIDED INDEXES AS A FUNCTION OF THE UNUSED CAPACITY OF L-TH LEVEL OF BALANCED INDEX

X Axis = unused capacity of L-th level of balanced index (percent)
Y Axis = relative search performance improvement (percent)



Graph 4.8: L = 3, Threshold level = 2, no-shuffle method.

The results of Graphs 4.2-4.10 were normalized in the sense that the relative performance improvement was plotted as a function of the unused capacity of an L-level balanced index. Those graphs do not indicate how much data may be stored in a sub-tree of height $L-1$ as compared to a balanced tree of height L. To address that, the ratio of the capacity of the portion of a lop-sided index that may be contained within a maximum depth of $L-1$ to the capacity of a L-level balanced index is plotted in Graph 4.11, and this ratio is referred to as the $(L-1)$-to-L ratio. Graph 4.11 is plotted as a function of the fullness of the L-th level of the balanced index, for various branching factors. This graph shows that the $(L-1)$-to-L ratio sharply decreases within the first 20% of the L-th level of the balanced index, and it decreases gradually from that point until the L-th level of the balanced index is filled. Graph 4.11 also shows the effect of the branching factor. The higher the branching factor, the $(L-1)$-to-L ratio is more highly skewed.

This graph graphically illustrates that lop-sided indexes are particularly well-suited to query distributions that obey the well-known 80–20 *rule*. The 80-20 rule of thumb is an approximation to realistic distributions that have been commonly observed in commercial applications [KNUT73]. This rule states that 80% of the queries access 20% of the database. The same applies to the most active 20% of the database, i.e., 64% of the queries access 4% of the database, etc. For query distributions that obey the 80-20 rule, a substantial portion of the most frequently accessed data may be maintained within a depth of $L-1$ levels in a lop-sided index.

The results in Graph 4.11 do not depend on the value of L, but rather only on the branching factor, b. The formula for computing the results in Graph 4.11 was:

$$\frac{(b-1)\times b^{L-2}}{b^{L-1}+f\times(b^{L}-b^{L-1})}$$

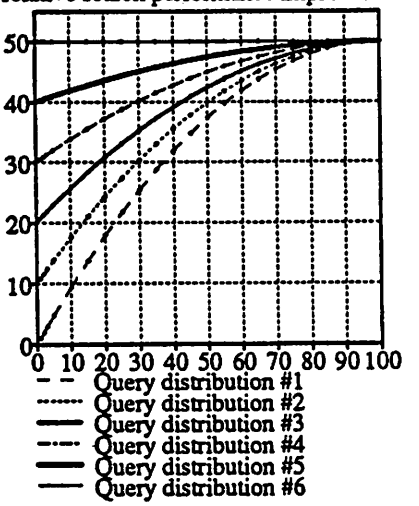
which reduces to:

$$\frac{1-\frac{1}{b}}{1+f\times(b-1)}$$

where f is the fraction of fullness of the L-th level of a balanced index.

RELATIVE SEARCH PERFORMANCE IMPROVEMENT OF LOP-SIDED INDEXES AS A FUNCTION OF THE UNUSED CAPACITY OF L-TH LEVEL OF BALANCED INDEX

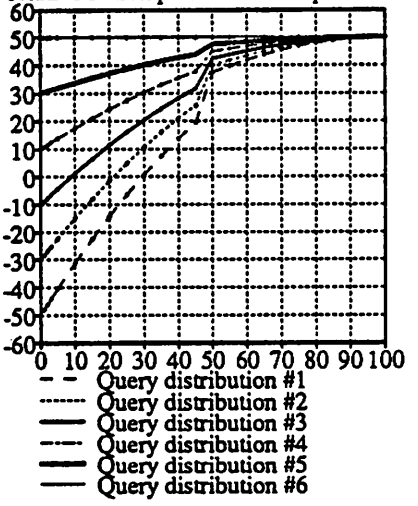
X Axis = unused capacity of L-th level of balanced index (percent)
Y Axis = relative search performance improvement (percent)



Graph 4.9: $L = 3$, Threshold level = 2, skeleton method.

RELATIVE SEARCH PERFORMANCE IMPROVEMENT OF LOP-SIDED INDEXES AS A FUNCTION OF THE UNUSED CAPACITY OF L-TH LEVEL OF BALANCED INDEX

X Axis = unused capacity of L-th level of balanced index (percent)
Y Axis = relative search performance improvement (percent)



Graph 4.10: $L = 3$, Threshold level = 2, shuffle method.

Graphs 4.12-4.14 show the fullness of the L-th level of a balanced index, as a function of the L-level balanced index capacity, for various branching factors. These graphs were plotted using a logarithmic scale, and the X-axes were labeled with the $\log_{10}$ of the balanced index capacities. Graphs 4.12-4.14 show the growth rates of L-level balanced indexes for values of L equal to 3, 4, and 5, respectively. These graphs illustrate that the growth rate is directly related to the branching factor, which follows from the fact that the growth rate is proportional to b^L, where b is the branching factor. Within each graph, however, the growth rates are linear since L is constant. The plots appear non-linear due to the horizontal logarithmic scale.

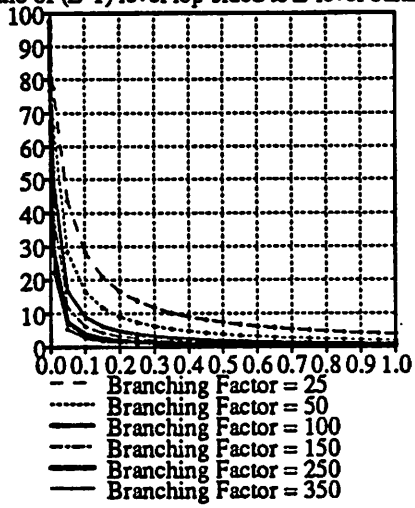
In the next section, generalizations to the lop-sided index are examined.

4.8. Generalizations and Areas of Future Research for Lop-Sided Indexes

Generalizations to the design of lop-sided indexes presented in the previous sections are possible. One important generalization would be self-adapting lop-sided indexes for non-uniform query and *data* distributions. Other topics for future research related to lop-sided indexes are (1) the issue of minimizing the impact of shuffling nodes (between or across levels) on insertion performance in lop-sided indexes that use the shuffle method, and (2) choosing the optimal number of partitions for determining the HFA and LFA nodes.

RATIO OF (L-1) LEVEL LOP-SIDED TO L LEVEL BALANCED INDEX CAPACITIES, AS A FUNCTION OF FULLNESS OF L-TH LEVEL OF BALANCED INDEX

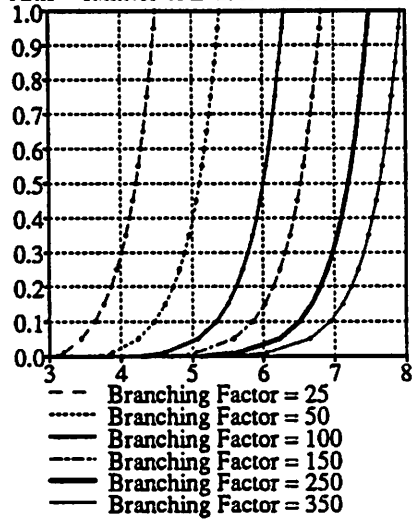
X Axis = fullness of L-th level of balanced index
Y Axis = ratio of (L-1) level lop-sided to L level balanced capacities



Graph 4.11

FULLNESS OF L-TH LEVEL OF BALANCED INDEX, AS A FUNCTION OF L LEVEL BALANCED INDEX CAPACITY

X Axis = log base 10 of L level balanced index capacities
Y Axis = fullness of L-th level of balanced index



Graph 4.12: $L = 3$

4.8.1. Adapting to Non-Uniform Data Distributions

Up to now, the assumption driving the design of lop-sided indexes has been that the query distribution is non-uniform and known in advance, and the data distribution is uniform. The case of the non-uniform *data* distribution may be further subdivided into two cases: (a) the data distribution is known in advance, and (b) the data distribution is not known in advance. Case (a) may be dealt with in a straightforward manner, since if both the data and query distributions are known in advance, the skeleton method may be used.

To deal with case (b), mechanisms for dynamic tree reorganization, such as the shuffling method presented earlier, can be applied directly to maintain the desired degree of lop-sidedness for any data distribution. Particularly in this case, the overhead of shuffling may be justified. The goal of minimizing this overhead is discussed in the next section.

4.8.2. Minimizing the Impact of Shuffling

The shuffle method lop-sided index "shuffles" a node from a higher-level high-frequency access non-leaf node to a lower-level low-frequency access non-leaf node. This may result in a series of shufflings from higher to lower level nodes, with intermediate *sideways shuffling* within a level, i.e., nodes may need to be moved from left-to-right (or vice versa) within a level in order to make room for nodes being shuffled down from a higher level. Sideways shuffling would occur in the event that all the nodes at the level which the first shuffled-down node was moved to were full. The worst case of *cascading shuffles* occurs when the data is inserted in increasing order, and the most frequently accessed data values are some percentage of the largest values. In that case, there will be a continuous cascading of nodes from the HFA to LFA nodes throughout the entire index. This scenario is not a contrived case. For

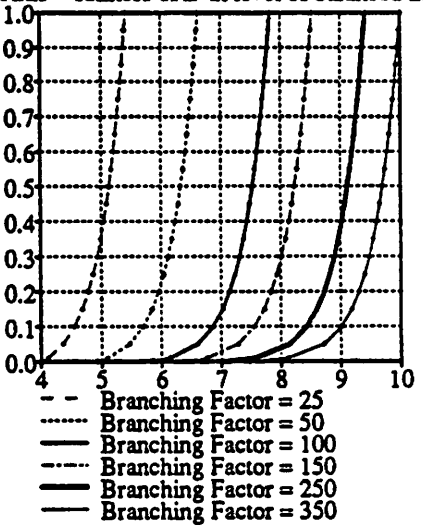
example, in the case of historical data, the data is inserted in time-sorted order, and the most likely query distribution is the one that most frequently accesses recent historical data. Therefore, the problem of the insertion performance impact caused by cascading shuffles needs to be addressed, or else a scheme that either avoids or minimizes shuffling must be devised. Although some form of shuffling seems inevitable in a dynamic lop-sided index, more research is required to investigate whether more clever schemes are possible.

4.8.3. Choosing the Optimal Number of Partitions

As discussed in the description of the performance study, the choice of the number of partitions parameter, *NPARTS*, used in the experiments was arrived at by empirical trials. This parameter controls the number of subdivisions of the indexed domain that are used in conjunction with the query probability distribution function for classifying nodes as being either high-frequency access or low-frequency access nodes. A systematic method for choosing the optimal number of partitions would be desirable. Such a method would have to take into account the query probability distribution function, the maximum and minimum values of the indexed domain, the index node cardinality, and the expected number of tuples.

FULLNESS OF L-TH LEVEL OF BALANCED INDEX, AS A FUNCTION OF L LEVEL BALANCED INDEX CAPACITY

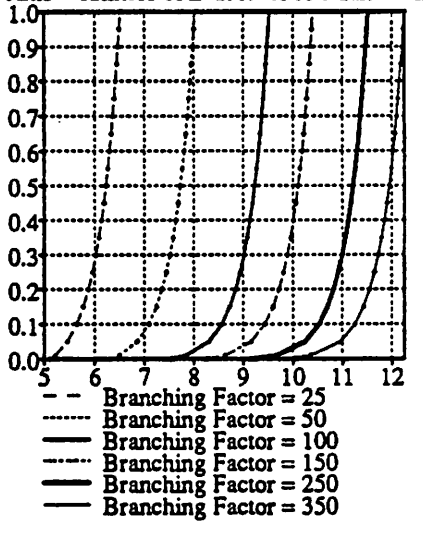
X Axis = log base 10 of L level balanced index capacities
Y Axis = fullness of L-th level of balanced index



Graph 4.13: $L = 4$

FULLNESS OF L-TH LEVEL OF BALANCED INDEX, AS A FUNCTION OF L LEVEL BALANCED INDEX CAPACITY

X Axis = log base 10 of L level balanced index capacities
Y Axis = fullness of L-th level of balanced index



Graph 4.14: $L = 5$

4.9. Summary and Conclusions

This chapter has presented the notion of lop-sided indexes to support queries that are non-uniform in the indexed domain. Large optical disk-based historical data archives have changed the assumptions of traditional database indexing by (1) vastly increasing the amount of data in a single database, and (2) the expected query distribution on historical data will usually be a highly skewed query distribution. These query distribution characteristics would be best served by lop-sided indexes, since they are well-suited to indexing a very large collection of data that are accessed in a non-uniform manner.

Simulations of lop-sided B+-Tree indexes were implemented, and the performance of the indexes was measured with respect to a corresponding balanced B+-Tree index counterpart. Over a range of non-uniform query distributions, lop-sided indexes provided a significant performance improvement over balanced indexes. Three methods were designed to either avoid or handle the potential problem of index reorganization: the methods were designated (1) no-shuffle, (2) skeleton, and (3) shuffle. The no-shuffle method avoids index reorganization as does the skeleton method, but the latter uses an estimate of the number of tuples to be indexed in order to reduce the likelihood that reorganization will be required. The shuffle method performs index reorganization, but has the undesirable potential problem of cascading shuffles, which would greatly reduce insertion performance. The simulation performance experiments showed that skeleton lop-sided indexes provided the best overall performance, followed by shuffle and then no-shuffle indexes.

Possible topics for future research in this area are: (1) lop-sided indexes which adapt to dynamically changing query distributions, i.e., to use query statistics to estimate future query distributions so that lop-sided indexes may adapt to recent

and possibly changing query distributions, as opposed to being restricted to static query distributions; and (2) to find ways to minimize or avoid the insertion performance impact of cascading shuffles.

CHAPTER 5

MIXED-MEDIA INDEXES

5.1. Introduction

The previous chapters have focused on Segment Indexes to improve spatial access methods and Lop-Sided Indexes for non-uniform query distributions. Non-uniform query distributions are particularly likely to occur when large historical data archives are maintained. In this chapter, *mixed-media indexes* are investigated. A mixed-media index is an index which may span magnetic and optical disk. With the advent of both write-once read-many (WORM) and rewritable (WMRM) optical disk technology, support for large historical data archives in a database management system has now become cost-effective. The motivation for a mixed-media index is that as data archives on optical disks become very large, so do their associated indexes. Storing the archive indexes entirely on magnetic disk may be too costly, while storing them entirely on an optical disk may be too inefficient, resulting from the limitations of data structures that may be maintained on a write-once medium and the slow access times of optical disks as compared to magnetic disks.

Although rewritable optical disks are presently available, WORM optical disks remain a desirable, and in some cases, an essential storage medium. For example, some banking and financial applications may require that an immutable audit trail be maintained. Other advantages of WORM optical disks over rewritable optical disks include: (1) WORM optical disks are less expensive than WMRM optical disks;

(2) WORM optical disks are available in very large capacities, e.g., 12 inch platters that store 6.4 Gb per platter, as compared to 5 inch WMRM optical disks that store 650 Mb per platter; (3) WORM optical disks have faster write transfer times by about a factor of 2, since WMRM optical disks require a separate *zero* write pass to clear the block contents to be overwritten. For these reasons, this chapter will primarily focus on indexing techniques for WORM optical disks, although the techniques and ideas apply equally well to WMRM optical disks.

Since WORM optical disk block contents may not be modified after their initial writing, the issue arises as to how best organize a historical database archive and associated indexes on such a write-once storage medium, given that the contents of the historical data archive are not known in advance and the archive grows incrementally over time.

5.1.1. Hypothesis

The hypothesis that motivated the set of experiments which are the subject of this chapter is that suitably designed *mixed-media* or *composite* index structures, i.e. indexes which may span magnetic and optical disk, will outperform an index structure that is contained entirely on optical disk in terms of search performance, and may approach or equal the performance of an index that is entirely contained on magnetic disk. The two principal advantages of allowing indexes for historical data relations to span magnetic disk (MD) and optical disk (OD) media, as opposed to being exclusively restricted to either medium, are:

(1) improved search and insert performance as compared to indexes that are completely contained on optical disk, and

(2) reducing the cost per bit of disk storage required for indexes as compared to storing historical data indexes entirely on magnetic disk.

This work investigates the performance of four indexing strategies for supporting historical database archives. A simulated workload was used to drive implementations of these strategies and produce performance statistics regarding indexes that were contained in each of three storage media environments: (1) a historical data index stored entirely on magnetic disk, (2) a historical data index stored entirely on optical disk, and (3) a historical data index that may span magnetic disk and optical disk.

The first environment was included in the tests to provide a basis of comparison with the performance of the mixed-media and OD-only schemes, and for that environment the original R-Tree index [GUTT84] was employed. The index structure selected for the second environment was based on the Allocation Tree index [VITT85]. For the third environment, two page movement policies for migrating historical database indexes from magnetic disk to an optical disk-based archive were implemented. These page movement policies were previously proposed in [STON87], and were utilized by the index structures for the third environment mentioned above which were two variations of the R-Tree index.

The remainder of this chapter proceeds as follows. Section 2 outlines two algorithms for migrating indexes on historical data relations from a magnetic disk to an optical disk. Section 3 presents the results of the performance tests which compared each of the four indexing techniques. Section 4 contains a summary and conclusions.

5.2. Vacuuming Algorithms for Indexes on Historical Data Relations

5.2.1. Definition: Data and Index Vacuuming

The term *vacuum* is defined as the transfer from magnetic to optical disk of pages containing either historical data records or index records that reference historical data records. Since the focus of this research is on indexing structures for

historical archives as opposed to the physical lay-out of the *data* contained in the archives, the use of the term *vacuum* in this chapter will specifically refer to the transfer of index pages from MD to OD.

5.2.2. Assumptions

There are many alternatives to managing current and historical data in a database management system. No assumptions are made regarding the particular underlying database management system in the performance experiments, other than to assume that current and historical data are each maintained in separate relations, referred to as *current data relations* and *historical data relations*, respectively. Updating or deleting a *current* data tuple results in a *historical* data tuple being appended to the historical data relation, whereas inserting a new tuple appends that tuple in a current data relation and has no effect on historical data relations. Current and historical relations may have separate and possibly different indexes associated with them. The justification for this is that there should not be an adverse impact on the performance of operations on current data relations introduced by the support of historical data relations. Also, the queries that are performed against the current and historical relations may be different, which would favor having different indexes on each of the relations. However, it may be advantageous to allow "recent" historical data to remain in the current data relation on magnetic disk until such time as a system process transfers a collection of such historical tuples to the historical relation on optical disk. Current data relations and their associated indexes reside on magnetic disk, whereas historical data relations are contained on optical disk. Given this storage architecture, queries on current data are satisfied by searching indexes on the current data relations, whereas queries on historical data may require searching indexes on both the current and historical data

relations.

5.2.3. Two Vacuuming Algorithms

Two index vacuuming algorithms which were proposed in [STON87] have been implemented for this study. Both of these algorithms produce variations of an R-Tree index which spans magnetic and optical disks. An R-Tree was chosen as a basis from which to design such indexes, since it provides fast access to multi-dimensional spatial data objects. As discussed in Chapter 1, historical data may be represented in a multi-dimensional space, in which *time* is one dimension. As a baseline comparison, the two R-Tree variations were compared to the original R-Tree contained on magnetic disk, which is hereafter referred to as the *Single Root MD R–Tree*, or more succinctly as the *MD–RT* index. The original R-Tree was presented in Chapter 2. The index structures compared in this study are described in the following sections.

The following two *MD/OD R–Tree* indexes are R-Trees which may span magnetic and optical disk, i.e., their nodes may be stored either on magnetic or optical disk.

5.2.4. Index MD/OD-RT-1: The Single Root MD/OD R-Tree Index

The Single Root MD/OD R-Tree Index is constructed from a standard R-Tree by the following simple vacuuming algorithm. Whenever the R-Tree index on magnetic disk reaches a threshold size near its maximum allotted size, the *Vacuum Cleaner Process* (VCP) moves some fraction of the *oldest* leaf pages (left-most leaf pages in the time domain, assuming the time domain is the horizontal dimension and increases to the right) to the archive. This fraction, *V_CANDIDATES*, is a tunable parameter which determines the fraction of leaf nodes that are migrated onto optical disk

storage per vacuuming operation. If the value of *V_CANDIDATES* is too high, some leaf nodes may be vacuumed that would otherwise receive new tuples and would thus lead to poor storage utilization. If the value of *V_CANDIDATES* is too low, each vacuuming operation may transfer only a small number of nodes, thus requiring a higher frequency of vacuuming. The optimal choice of *V_CANDIDATES* is a complex optimization problem which depends on the width (in the time dimension) of the *active insertion window*, which is defined below. Historical data is inserted in approximately time-sorted order, and nodes that fall within the active insertion window are continuing to receive historical data, as opposed to nodes which contain older historical data and are therefore out of the active insertion window, which are said to be *inactive*. The width of the active insertion window is a function of the rate at which the time attributes of the inserted data increase with respect to the vacuuming frequency, and is bounded by the magnetic disk space allotted to the index. In the performance experiments reported in this chapter, *V_CANDIDATES* was set to 25%, i.e., up to 25% of the oldest leaf nodes on magnetic disk were candidates for vacuuming during each index vacuuming operation of the VCP. This value of *V_CANDIDATES* was chosen as a result of empirical tests.

Following the vacuuming of the leaf nodes, the VCP then vacuums all of the ancestor nodes of the vacuumed nodes that point entirely to nodes on the archive. This non-leaf node vacuuming is applied recursively to all of the higher level nodes, except to the root node which is never a candidate for vacuuming.

An example of a MD/OD-RT-1 index is illustrated in Figure 5.1. This figure shows an R-Tree that has each of its nodes marked with either an M to signify that the node resides on MD, or an O to mean that it resides on OD. As this figure illustrates, a substantial portion of the nodes that contain the *oldest* data (i.e., *leftmost* in the time domain, assuming the time domain is the horizontal axis and increases to

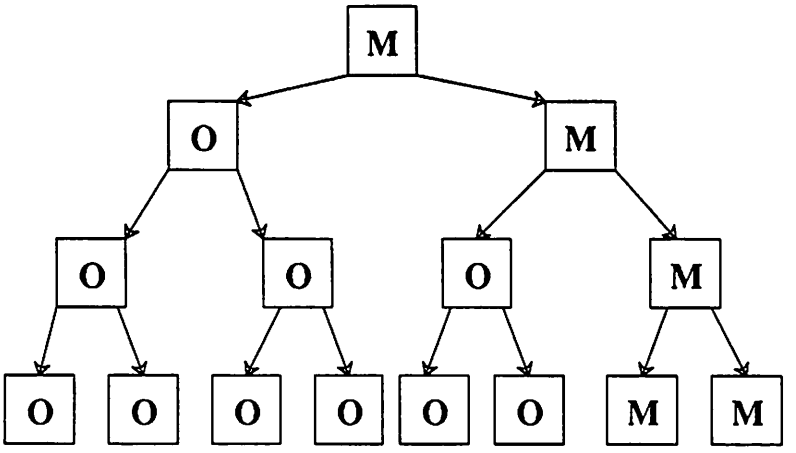


Figure 5.1: Single Root MD/OD R-Tree

the right) may reside on OD.

5.2.5. Index MD/OD-RT-2: The Dual Root MD/OD R-Tree Index

The Dual Root MD/OD R-Tree Index consists of a pair of R-Tree indexes, both rooted on magnetic disk. The first R-Tree is contained completely on magnetic disk, and the second is rooted on magnetic disk and has its lower levels on optical disk. The Dual Root MD/OD R-Tree Index is constructed from a standard R-Tree by the following vacuuming algorithm. The VCP is invoked when the size of the first R-Tree index on magnetic disk reaches its maximum allotted size. When first invoked, the VCP vacuums all of the first R-Tree's nodes, except its root node, to the optical disk and allocates a root node on magnetic disk for the second R-Tree. Then, a new R-Tree is constructed on magnetic disk as new historical data index records are inserted into the index. Subsequently, each time the VCP is invoked, it vacuums all of the first R-Tree's nodes except the root node, and inserts the immediate descendants of the first root into the corresponding level of the second R-Tree on magnetic

disk. As more vacuuming operations occur, the number of magnetic disk nodes of the second R-Tree will increase, due to conventional R-Tree node splitting which may propagate up to the root node. Over time, there would continue to be two R-Trees. The first would be completely on magnetic disk and periodically archived. Insertions are made to the first R-Tree while searches are performed by descending both R-Trees.

An example of a MD/OD-RT-2 index is illustrated in Figure 5.2. In that figure, the *first R–Tree* is entirely on MD, and may grow to a finite size, at which time it is vacuumed to OD, and merged into the *second R–Tree* which has its lower levels residing on OD and its root node on MD.

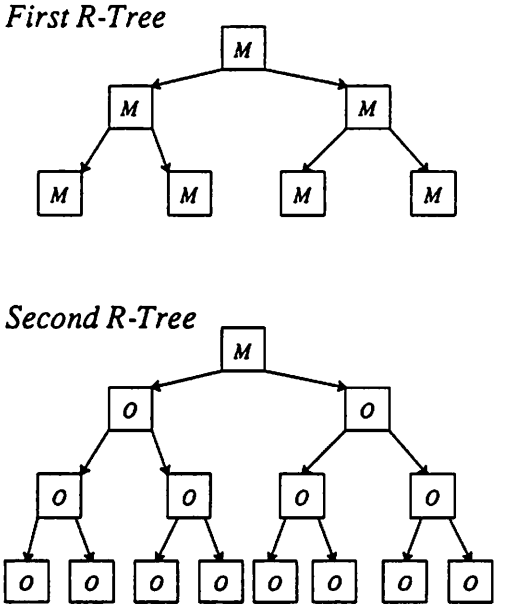


Figure 5.2: Dual Root MD/OD R-Tree

5.2.6. Index OD-AT: The Allocation Tree Index

The version of Allocation Trees implemented for this performance study is the first of the two types that Vitter described in [VITT85], which is similar to a data structure developed by Rathmann [RATH84]. Allocation Tree records are inserted in a breadth-first manner, and the tree is searched in a depth-first manner. A simple characterization of Allocation Trees is a linked list of increasingly deeper depth-first search trees, where each successive search tree is one level deeper than its predecessor in the list.

An example of an Allocation Tree with a non-leaf node branching factor $b = 3$ is illustrated in Figure 5.3. In that figure, the nodes represent allocated OD pages, and the arrows are the index records that point to other allocated pages. This structure was primarily designed for efficiently locating the most recently allocated page on a

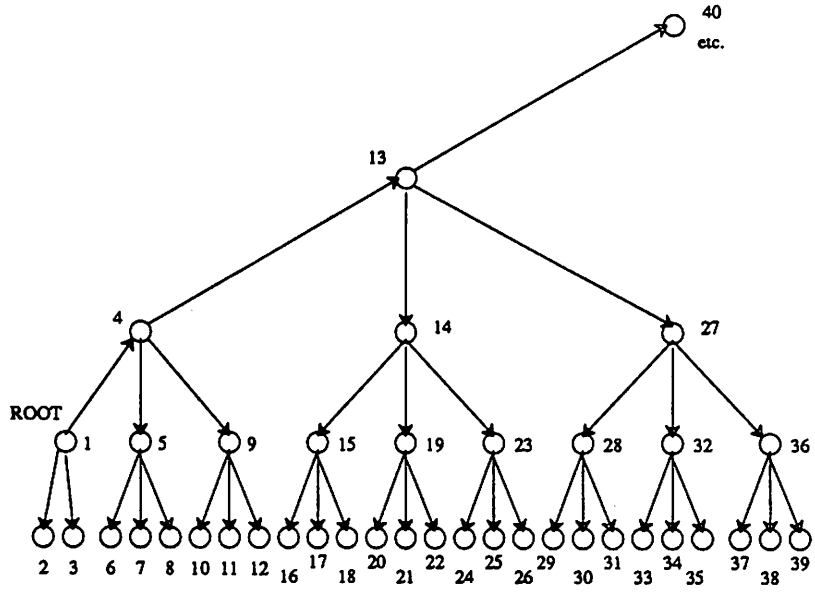


Figure 5.3: Allocation Tree

write-once medium. Starting with a pointer to node 1, the most recently allocated page may be retrieved with approximately $2 \times \log_b(n)$ page accesses, where n is the number of allocated pages and b is the non-leaf node branching factor. This point is evident in Figure 5.3, which shows that in the worst case the tree traversal from root 1 to the most recently allocated node must first *climb* $\log_b(n)$ nodes to reach the *highest* node (along the top of the structure illustrated in Figure 5.3), and then must *descend* $\log_b(n)$ nodes to reach the most recently allocated leaf node.

The Allocation Tree requires either that fractions of an OD page may be written, or else pages that have not yet had their index records filled in (the *fringe* of the most recently allocated non-leaf pages which do not yet point to other OD pages) are buffered in main memory and on MD until they are completely filled. Since most if not all WORM ODs do not allow partial writing to a page, unfilled pages will usually have to be buffered, thus making the supposed OD-only Allocation Tree really another case of an MD/OD index, albeit with only a small fraction of pages on MD. For the purposes of this study, Vitter's "partial OD page write" assumption was adopted so that the Allocation Tree would qualify as an OD-only index.

Allocation Trees store data records within the non-leaf nodes of the index structure. R-Trees, on the other hand, may store either actual data or else data page pointers in its leaf nodes depending on whether the index is being used as a primary or secondary index, respectively. In order to make meaningful comparisons of the performance characteristics of these two indexes, both indexes must store either data records or data page pointers. In the performance experiments the latter was done, thus making both the Allocation Trees and R-Trees secondary indexes. Therefore, the Allocation Trees had data page pointers, hereafter referred to as *data records* of 32 bytes (two 4-byte words for Tmin and Tmax, one 4-byte tuple identifier,

and 20 bytes for a key descriptor and value)[1]. These are to be distinguished from *index records* which point to other Allocation Tree nodes on the archival optical disk medium.

All leaf nodes in Allocation Trees contain data records only, whereas all non-leaf nodes may contain both data records and index records. Thus, in this sense Allocation Trees are unlike B-Trees or R-Trees, both of which store index records in non-leaf nodes, and data records in the leaf nodes. In Allocation Trees, any proportion of data to index records in the non-leaf nodes is permitted, except that there must be at least two index records per non-leaf page[2]. Five percentages of index records in the non-leaf nodes were used in the performance tests: 2%, 25%, 50%, 75%, and 100%.

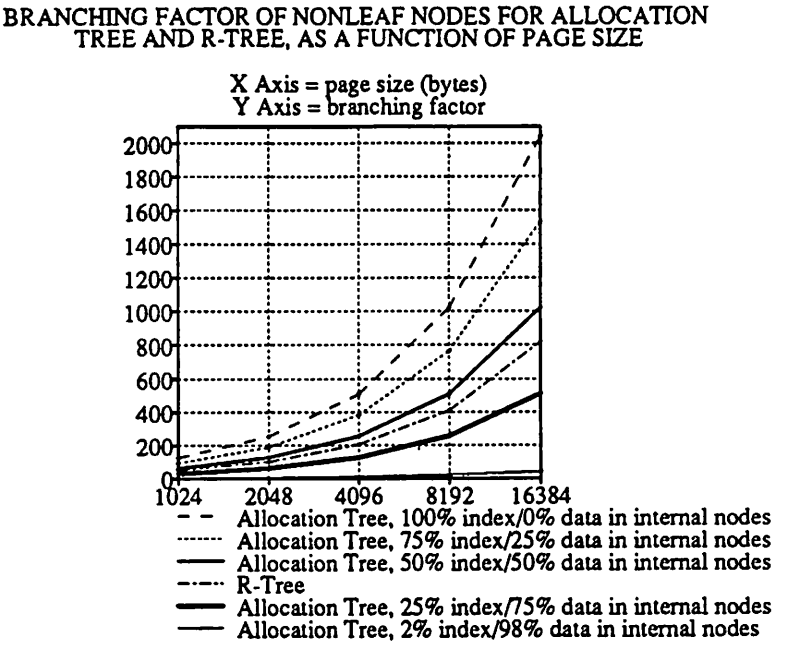
5.2.7. Branching Factors

Five page sizes were tested for each of the indexing structures. Logical pages were assumed to be physically contiguous on one track on either the magnetic or optical disks, and therefore could be read in one I/O operation on either type of disk. The maximum possible branching factors (the maximum number of index records per non-leaf page) for the Allocation Tree and R-Tree indexes are shown in Graph 5.1.

---

[1]A 32 byte data record was also used in the R-Tree experiments, so that the search performance comparisons between R-Trees and Allocation Trees would be fair.

[2]Strictly speaking, at least one is required, but a minimum of two provides reasonable search performance, since otherwise the index degenerates into a linked list.



Graph 5.1: Branching Factors of R-Tree and Allocation Tree Non-Leaf Nodes

5.3. Performance Experiments

The performance experiments were carried out by constructing indexes for two collection of rectangles which were created using a random number generator for both the time domain (*Tmin* and *Tmax*) and data fields (*Vmin* and *Vmax*). These synthetic data relations were constructed so that the time domain intervals were approximately sorted with respect to the time domain, and that the data interval values were not correlated with the time interval values. This was done in order to model the characteristics of a historical data relation. In the experiments, it was assumed that the historical data was inserted in order of its *Tmax* fields, where *Tmax* represents the time that the transaction which either deleted or modified an existing tuple (which causes the former value of the tuple to be inserted into the historical

archive) committed.

The number of index records generated ranged from 30,000 to 100,000. By convention, the horizontal rectangle coordinates represented values in the time domain, and the vertical coordinates represented some interval of a data (i.e., non-temporal) domain. In the case of both the R-Tree and Allocation Tree indexes, for the first data relation, the time range intervals were generated as follows. $Tmin_0$ and $Tmax_0$ were initialized to zero, and then each time range interval limit pair, $Tmin_i$ and $Tmax_i$, were separately incremented from its predecessor by a random value uniformly distributed over $[10^3, 10^4]$ while satisfying the constraint that $Tmin_i < Tmax_i$. For each rectangle, the other two (vertical) coordinates, $Vmin_i$ and $Vmax_i$, were generated by assigning each a random value from a uniform distribution over [0, 2.0e+09], such that $Vmin_i < Vmax_i$. Therefore, for the first data relation, the domain of the indexed data was [0, 2.0e+09] in the vertical dimension, and [0, $5500 \times nrecords$] in the horizontal dimension, where *nrecords* was the number of records inserted, and the average increment of the time dimension per inserted record was: $((10^4 - 10^3) / 2) + 10^3 = 5500$.

For the second data relation, the time range intervals $Tmin_i$ and $Tmax_i$ were initialized to zero and each was separately incremented from its predecessor by a random value uniformly distributed over $[10^4, 3 \times 10^4]$ such that $Tmin_i < Tmax_i$. For each rectangle, the vertical (data) coordinates, $Vmin_i$ and $Vmax_i$, were generated by assigning each a random value from a uniform distribution over $[0, 10^5]$, such that $Vmin_i < Vmax_i$. Therefore, for the second data relation, the domain of the indexed data was $[0, 10^5]$ in the vertical dimension, and $[0, 2 \times 10^4 \times nrecords]$ in the horizontal dimension, as the average increment of the time dimension per inserted record was: $((3 \times 10^4 - 10^4) / 2) + 10^4 = 2 \times 10^4$.

Two-dimensional R-Trees may be used to index rectangles, lines, or points, since either dimension may index *interval* or *point* data. Interval indexes in two dimensions are required for indexing rectangle data. Historical indexes on time intervals and point data is likely to be a common variety in practice. For example, an index on salary histories of employees would be an index over a collection of historical point data, where the points represented employee salaries. Such an index could be used to efficiently satisfy queries such as:

```
retrieve (EMP.name)
        using EMP[1988, 1989]
        where EMP.salary = 10000
```

to find all the names of employees who were active at some time within the time interval represented by [1988, 1989] and whose salary was $10,000.

Another variety of historical data may consist of a historical archive of interval *data*. Such a data collection would require an index over interval data in both the time and some other (non-temporal) dimension. In this study, intervals in two dimensions, i.e. rectangles, were chosen as the data type for indexing, because such spatial data poses a greater challenge to any indexing structure, and the case of indexing intervals in one dimension and points in the other dimension is a special case of indexing interval data in two dimensions.[3] The following query provides an example in which an index on historical interval data would be useful.

```
retrieve (CITY.name)
        using CITY[21 June 1990, 23 September 1990]
        where CITY.diurnal_temperature_range
        overlaps [90° F, 100° F]
```

---

[3] The choice of interval data over point data for the non-temporal attribute did not have a significant effect on the experimental results, since the dominant features of the data relations were that the temporal data was approximately sorted and the non-temporal data was not correlated with the temporal data.

This query finds the names of all cities whose diurnal (daily) temperature range ever fell within 90° to 100° F during the Summer of 1990.

Each of the four index types described above were compared in terms of their search and insert performance, as well as index space requirements. To measure search performance, a large number of random interval searches were generated over the entire data domain contained in the index.

Each performance test consisted of two phases, an *insert phase* and a *search phase*. Each test program began with an empty index. During the insert phase, an input data file was read and an index was constructed for that data. Insert performance was measured for the last 10% of the records, when the index was nearly its final size. During the search phase, a series of 100 random searches was performed. The following queries were used to generate the random search query sequences (in SQL):

Query 1:

```
select *
from test_relation
where tmax >= rand_time_min
      and tmin <= rand_time_max;
```

Query 2:

```
select *
from test_relation
where tmax >= rand_time_min
      and tmin <= rand_time_max
      and vmax >= rand_val_min
      and vmin <= rand_val_max;
```

In both queries, tmin, tmax, vmin, and vmax were attributes of test_relation. The rand_time_min/max and rand_val_min/max correspond to randomly generated search intervals, respectively. Query 1 generated a random search interval and Query 2 generated a random search "rectangle", i.e. a two-dimensional search space composed of two search intervals. The random search intervals were generated by

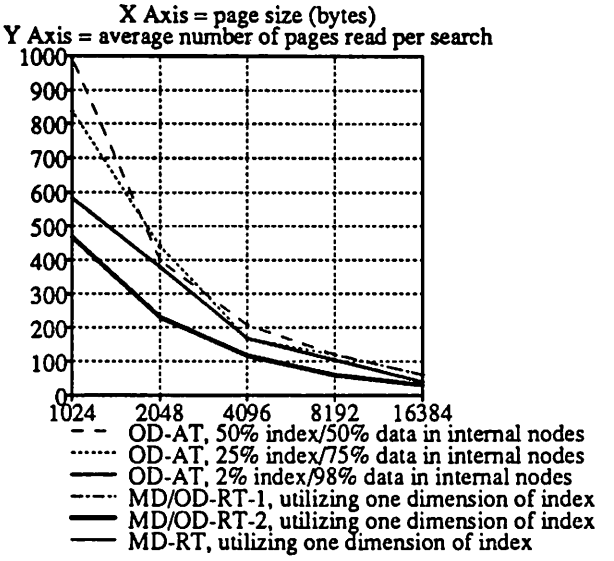
creating a random interval with a length that was uniformly distributed over the range [0, 50%] of the difference of the maximum and minimum values of the index records in each dimension. Each query was repeated 100 times using different random search arguments, and each sequence of searches was repeated for all of the index types. This series of 100 query executions retrieved approximately 20% and 10% of the index records when processing Query 1 and Query 2, respectively.

Four sets of experiments were performed, two each for data relation 1 and data relation 2. In the first set, the page size of the index was varied, using sizes of 1, 2, 4, 8, and 16 Kb, where in each of these tests the number of index records was 50,000. In the Allocation Tree tests, for each page size the percentage of space in the non-leaf nodes used for index records (%IR) was varied over 2%, 25%, and 50%. Values of %IR greater than 50% were not used in the experiments because the search performance of the Allocation Trees diminished as the %IR was increased. In the second set of tests, the number of index records was varied while using a page size of 1024 bytes. These tests were repeated with indexes consisting of from 30,000 to 100,000 records, in increments of 10,000 records. The third and fourth sets corresponded to the first and second, except that the input data was from data relation 2 rather than data relation 1. In the tests involving the MD/OD R-Tree indexes, the size of the magnetic disk area usable for the index was 256 Kb. The choices for the number of records and the size of the magnetic disk area usable for the index were dictated by the limited amounts of magnetic disk space that were available for conducting the experiments. Although these parameter values may be small with respect to "realistic" historical data indexes, they were large enough for analyzing the performance characteristics of the indexes.

A subset of the performance experiments reported in the following two sections were originally presented in [KOLO89]. The implementation of the R-Tree variations

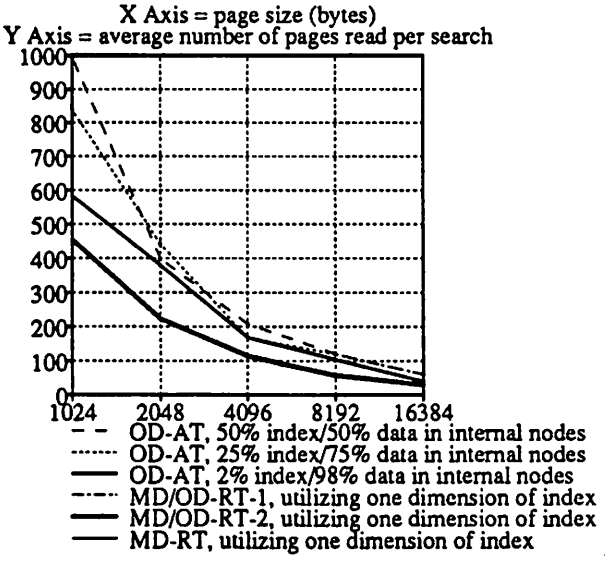
was later improved and those original experiments were repeated using the new R-Tree code. In addition, the experiments were expanded for this chapter to include the 2% and 25% index record tests of the Allocation Tree, and the entire set of experiments was repeated for data relation 2.

AVERAGE NO. OF PAGES READ PER SEARCH EXECUTING QUERY 1
USING ALLOCATION TREE AND R-TREE, AS A FUNCTION OF PAGE SIZE



Graph 5.2: number of records = 50,000; data relation 1.

AVERAGE NO. OF PAGES READ PER SEARCH EXECUTING QUERY 2
USING ALLOCATION TREE AND R-TREE, AS A FUNCTION OF PAGE SIZE



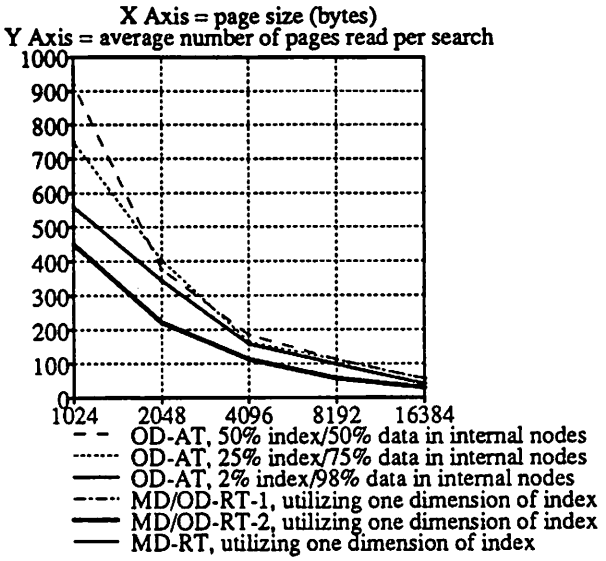
Graph 5.3: number of records = 50,000; data relation 1.

5.3.1. Performance Results: As a Function of the Page Size

The set of graphs to be discussed first are all plotted as a function of the page size. Graphs 5.2 and 5.3 show the average number of index pages read per search to find all the qualifying index records, for all of the R-Tree variations and for Allocation Tree indexes which had 2%, 25%, or 50% of the non-leaf node space used for index records.

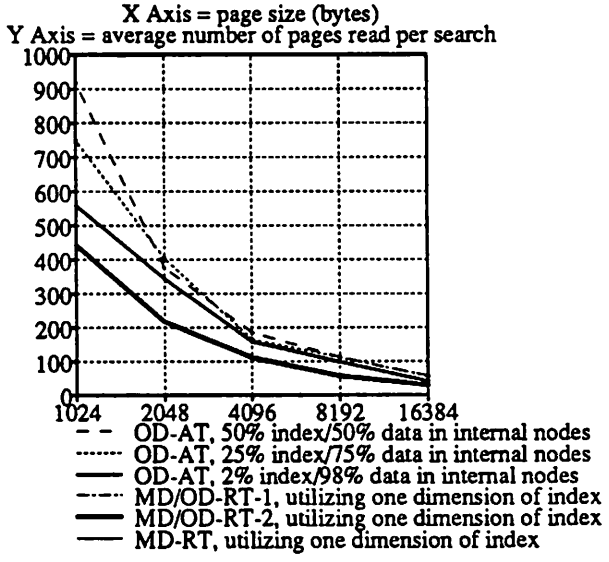
The curves in Graphs 5.2 and 5.3 show the performance data collected for one-dimensional and two-dimensional searches performed on data relation 1 corresponding to Query 1 and Query 2, respectively. These graphs show the performance of each of the indexes in terms of the average number of pages accessed per search, as a function of the page size for a database consisting of 50,000 records. The curves for the Allocation Tree indexes are the same in both graphs, since the same number of index pages were read when an Allocation Tree was used as the access path to execute either Query 1 or 2, i.e., all the records that satisfied the given time range search argument were examined when executing either query. The curves for the first two R-Tree variations (MD-RT and MD/OD-RT-1) were identical, and the curve for the third R-Tree variation (MD/OD-RT-2) was very close to the first two, so much so that all three appear as one thick line.

AVERAGE NO. OF PAGES READ PER SEARCH EXECUTING QUERY 1
USING ALLOCATION TREE AND R-TREE, AS A FUNCTION OF PAGE SIZE



Graph 5.4: number of records = 50,000; data relation 2.

AVERAGE NO. OF PAGES READ PER SEARCH EXECUTING QUERY 2
USING ALLOCATION TREE AND R-TREE, AS A FUNCTION OF PAGE SIZE



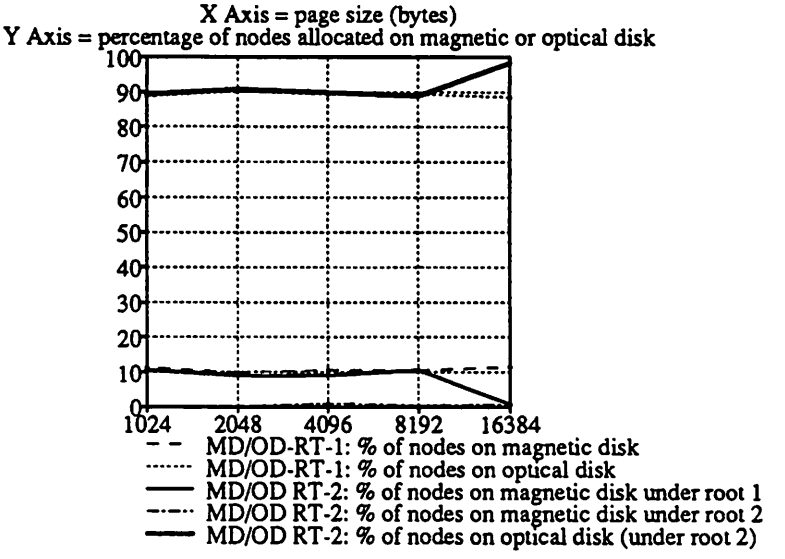
Graph 5.5: number of records = 50,000; data relation 2.

The curve for the R-Tree performance was slightly lower in Graph 5.3 In all of the search performance graphs, all three of the R-Tree curves were essentially equal and were lower than the curves for the Allocation Trees. The fact that the three R-Tree variations provided nearly the same performance is not surprising, and was related to both the order of insertion and the vacuuming frequency and quantity. Since the data was inserted in time-sorted order and only a small fraction of the "oldest" nodes on magnetic disk were vacuumed at a time, all of the insertions were performed onto MD resident pages. For the case of MD/OD-RT-1, inserts were not attempted on pages that had been vacuumed to OD, since those pages contained "old" data and hence were *inactive* with regard to receiving present or future inserted data. For the case of MD/OD-RT-2, the historical data R-Tree evolved into an index that had nearly identical search characteristics as the original MD-RT R-Tree that was entirely on MD. (for Query 2) than in Graph 5.2 (Query 1), because the second dimension of the index provided a slight reduction in the average number of pages accessed. The difference between the R-Tree performance between Queries 1 and 2 shown in Graphs 5.2 and 5.3, respectively, was very slight due to the fact that the data caused the higher level node regions to have good search resolution in the (horizontal) time domain, and poor resolution in the (vertical) data domain. That is, the regions of the higher level nodes tended to be very "tall and skinny", i.e., with respect their domains, the average high level node regions were short in the horizontal domain and long in the vertical domain. This was due to the insertion order of the data, which was sorted in the temporal dimension and random in the non-temporal dimension.

In Graphs 5.2 and 5.3, the three Allocation Tree variations performed with approximately the same performance, except at a page size of 1 Kb. At the page size of 1 Kb, the best search performance was inversely proportional to the fraction of

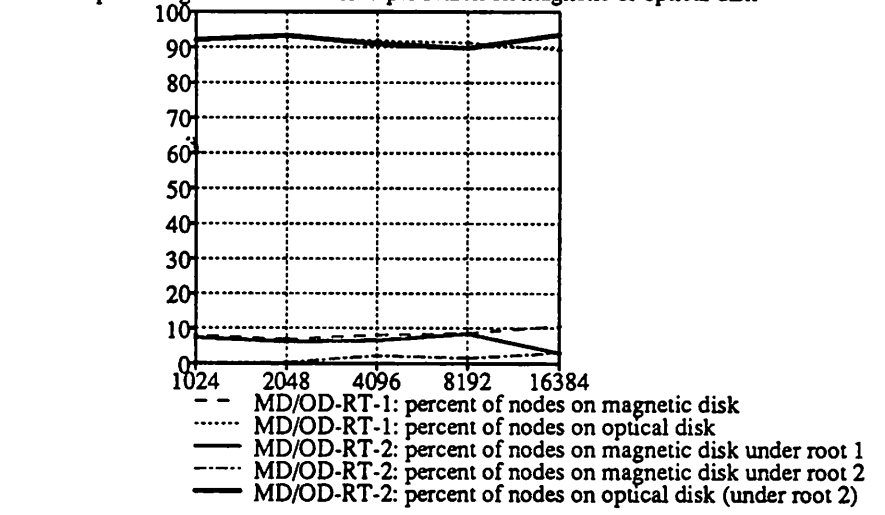
non-leaf node entries used for index records, i.e., the best performance was provided by the 2% index records per non-leaf node index, followed by the 25% and then 50% indexes, respectively.

PERCENTAGE OF NODES ALLOCATED ON MAGNETIC
OR OPTICAL DISK, AS A FUNCTION OF PAGE SIZE



Graph 5.6: number of records = 50,000; data relation 1.

PERCENTAGE OF NODES ACCESSED PER SEARCH ON MAGNETIC
OR OPTICAL DISK FOR QUERY 1, AS A FUNCTION OF PAGE SIZE

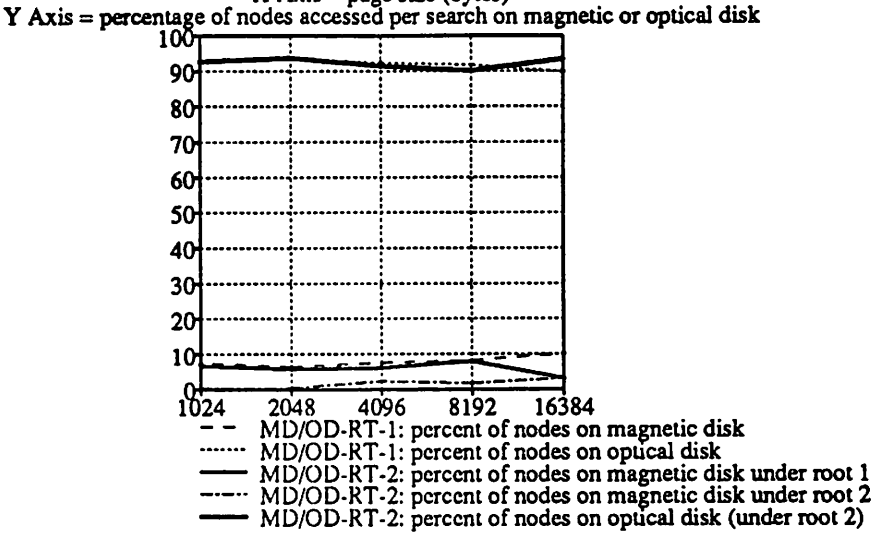


Graph 5.7: number of records = 50,000; data relation 1.

This result is not so straightforward, however. On the one hand, by storing a greater amount of data in non-leaf nodes, fewer page accesses are necessary to satisfy queries. On the other hand, a very low degree of index records in non-leaf nodes reduces the fanout, thus degenerating the index into a list structure as opposed to a tree. Since the Allocation Tree idea is basically to construct a list of trees, excessive reduction of the fanout transforms the index from a list of trees into a list of lists. As was shown in Graph 5.1, the branching factor (fanout) of the 2% index record Allocation Tree is quite low, compared to the other indexes.

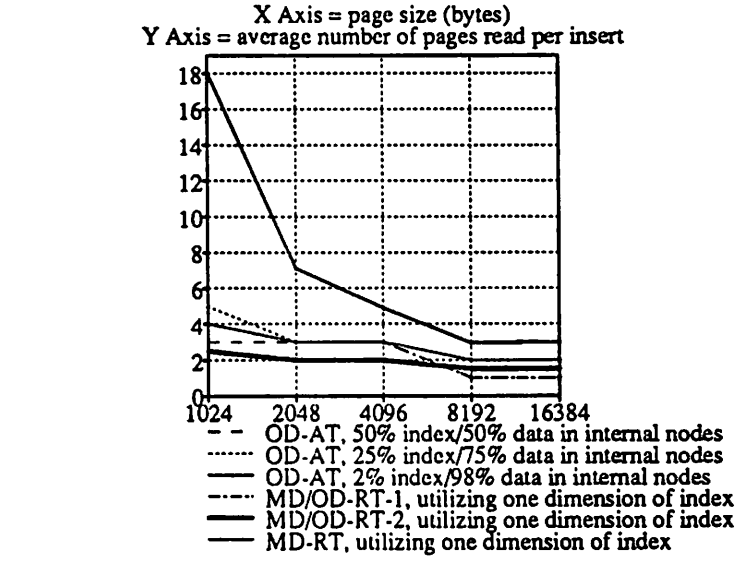
Graphs 5.4 and 5.5 correspond to Graphs 5.2 and 5.3, except that the indexed data was data relation 2. Graphs 5.4 and 5.5 are strikingly similar to Graphs 5.2 and 5.3, i.e., the rankings among all of the index types remained the same, and differed only in that the average number of pages accessed per search was slightly less in Graphs 5.4 and 5.5 than in Graphs 5.2 and 5.3, and the Allocation Trees had a greater decrease in page accesses as compared to the R-Trees. The domain of data relation 1 was characterized by a "long" data span and a "short" time span, whereas data relation 2 had a short data span and a long time span. Furthermore, the average time interval span per data record was relatively short in data relation 1 and long in data relation 2. Since the R-Trees had better time resolution than data resolution, and the Allocation Trees had only time resolution, by reducing the data span of the indexed data, and hence, the random queries which were generated, it is to be expected that both indexes would perform better than in the experiments involving data relation 1.

PERCENTAGE OF NODES ACCESSED PER SEARCH ON MAGNETIC OR OPTICAL DISK FOR QUERY 2, AS A FUNCTION OF THE PAGE SIZE



Graph 5.8: number of records = 50,000; data relation 1.

AVERAGE NO. OF PAGES READ PER INSERT USING ALLOCATION TREE AND R-TREE, AS A FUNCTION OF PAGE SIZE



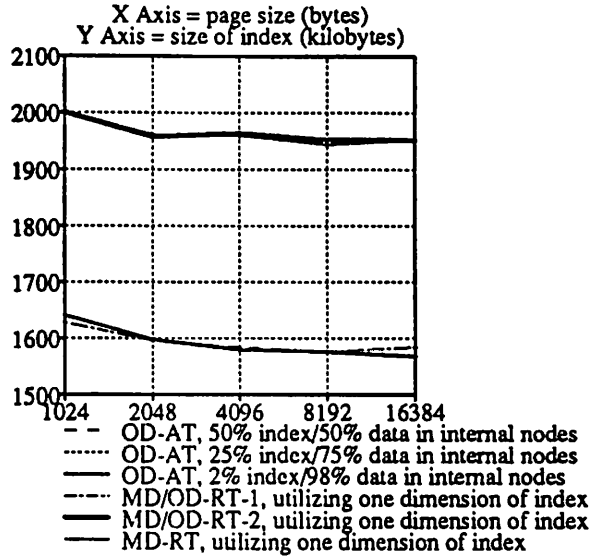
Graph 5.9: number of records = 50,000; data relation 1.

The Allocation Trees had a greater improvement than the R-Trees in the data relation 2 trials because the R-Trees had nodes which overlapped in the time domain, whereas the Allocation Trees had no overlap. The remaining graphs in this section are from the tests that involved data relation 1, since the results from data relation 2 were similar.

Graphs 5.2 through 5.5 compare the indexes in terms of absolute numbers of disk accesses, i.e., each magnetic or optical disk access was counted as one access, and no weighting factor was assigned to either type of access, though the access speeds of optical disks may be slower than that of magnetic disks by a factor of 3 to 7 [HP89, LMSI87]. Since the Allocation Trees were wholly contained on optical disk and the MD-RT R-Tree was wholly contained on magnetic disk, these indexes performed disk accesses that were exclusively on optical or magnetic disk, respectively. For each of the mixed-media R-Trees, Graph 5.6 shows the percentage of pages that were allocated on magnetic and optical disk. In Graph 5.6, the percentage of magnetic and optical disk pages are shown for the single root MD/OD R-Tree. For the dual root MD/OD R-Tree, the percentage of magnetic disk pages is broken down between the magnetic disk pages under each of the two roots. This graph shows that the percentage of optical disk nodes for both indexes remained fairly constant at approximately 90%. In the dual root MD/OD R-Tree with a page size of 16 Kb, the percentage of optical disk pages increased to close to 100%. Since the data contained in this graph was gathered only after the entire index was built, it was to be expected that the curves would show some variation depending on the amount of data inserted since the last vacuuming. In the dual root MD/OD R-Tree with a page size of 16 Kb, the last vacuuming occurred just before the index reached its final size, thus leaving a small percentage of nodes on MD. Also in the dual root MD/OD R-Tree, the percentage of MD nodes under root 2 was very small in all cases,

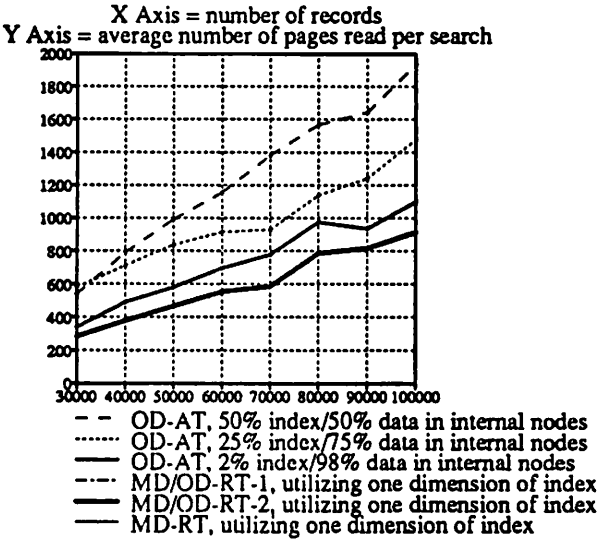
showing that the MD portion of the R-Tree only required a small amount of MD space.

SIZE OF ALLOCATION TREE AND R-TREE INDEXES, AS A FUNCTION OF PAGE SIZE



Graph 5.10: number of records = 50,000; data relation 1.

AVERAGE NO. OF PAGES READ PER SEARCH EXECUTING QUERY 1 USING ALLOCATION TREE AND R-TREE, AS A FUNCTION OF THE NO. OF RECORDS



Graph 5.11: page size = 1024 bytes; data relation 1.

Graphs 5.7 and 5.8 show the percentage of the average number of MD and OD pages accessed per search for each of the mixed-media R-Trees for Queries 1 and 2, respectively. These graphs are similar to Graph 5.6, except that the OD curves are slightly higher, as is the curve for the MD portion of the dual root MD/OD R-Tree. These differences reflect that most of the queries were satisfied by OD nodes, since about 90% of the data resided on OD, and the queries were uniformly distributed.

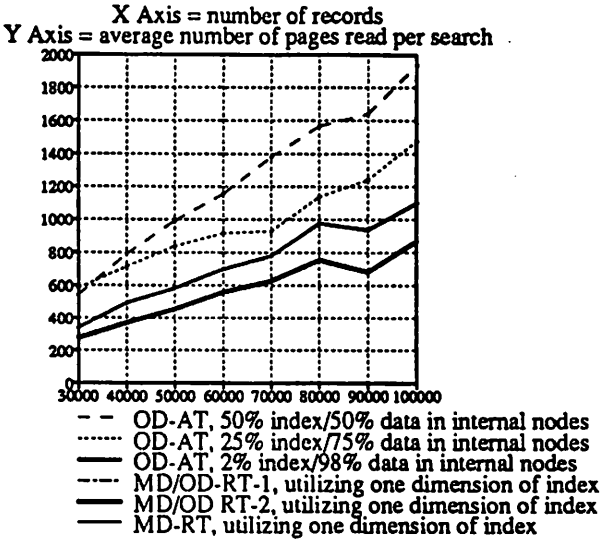
Graph 5.9 shows the average number of pages read per insert using each of the index types. Although search performance is considered paramount to insert performance, these results are important because they show that the performance of the Allocation Tree with 2% of its non-leaf node entries used for index records had very poor insert performance, particularly when the page size was less than 8 Kb. These statistics were collected during the last 10% of the insertions, when the index was nearly its final size. Since the 2% index record Allocation Tree had a low branching factor, many pages had to be accessed in order to add new records when the index grew from 45000 to 50000 records.

Graph 5.10 shows the size of each of the indexes in units of Kb. The R-Trees were approximately 23% larger than the Allocation Trees. However, R-Trees provided substantial search performance improvement, as was made evident by Graphs 5.2 through 5.5 for several page sizes.

5.3.2. Performance Results: As a Function of the Number of Records

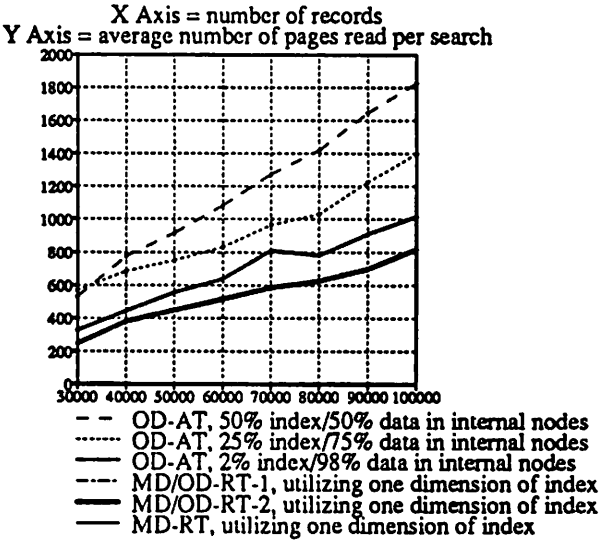
In the second set of experiments, the page size was fixed at 1 Kb, and the number of records was varied from 30000 to 100000, in increments of 10000.

AVERAGE NO. OF PAGES READ PER SEARCH EXECUTING QUERY 2 USING ALLOCATION TREE AND R-TREE, AS A FUNCTION OF THE NO. OF RECORDS



Graph 5.12: page size = 1024 bytes; data relation 1.

AVERAGE NO. OF PAGES READ PER SEARCH EXECUTING QUERY 1 USING ALLOCATION TREE AND R-TREE, AS A FUNCTION OF THE NO. OF RECORDS



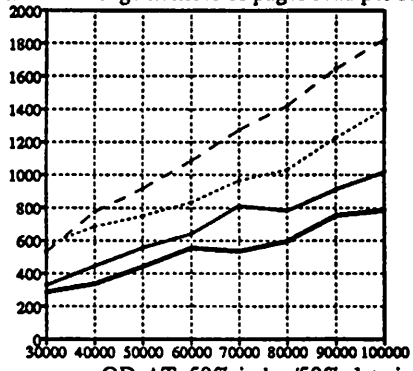
Graph 5.13: page size = 1024 bytes; data relation 2.

The search performance results for Queries 1 and 2 are given in Graphs 5.11 and 12 for data relation 1 and Graphs 5.13 and 5.14 for data relation 2, respectively. There are five similarities to the results shown in Graphs 5.2 through 5.5. (1) The curves for the Allocation Trees for both queries are the same. (2) The difference between the R-Tree curves between the two queries for each data relation are slight, where the curves from Query 2 are marginally lower than those from Query 1. (3) The three R-Tree variations had approximately the same search performance in terms of the average number of pages accessed per search. (4) The R-Tree variations outperformed the Allocation Tree variations, the latter of whose rankings were the 2%, 25%, and 50% index record per non-leaf node variations. (5) The curves from the experiments involving data relation 2 were slightly lower than those for data relation 1. The remaining graphs in this section pertain to the experiments involving data relation 1, as the results from data relation 2 were similar.

Graph 5.15 shows the percentage of MD and OD pages for each of the two mixed-media R-Tree indexes. Again, depending on the time of last vacuuming before the index reached its final size, the MD and OD portions fluctuated around 10% and 90%, respectively. Graphs 5.16 and 5.17 show the percentages of the average number of MD and OD pages accessed per search for each of the mixed-media R-Trees for Queries 1 and 2, respectively. These graphs show that the majority of the accesses were to OD nodes, since the OD curves are higher in these graphs than in Graph 5.15. This is to be expected, since 90% of the nodes were on OD, and the query distributions were uniform.

AVERAGE NO. OF PAGES READ PER SEARCH EXECUTING QUERY 2 USING ALLOCATION TREE AND R-TREE, AS A FUNCTION OF THE NO. OF RECORDS

X Axis = number of records
Y Axis = average number of pages read per search

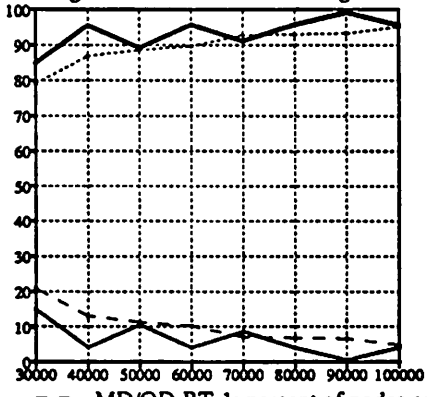


- - OD-AT, 50% index/50% data in internal nodes
····· OD-AT, 25% index/75% data in internal nodes
—— OD-AT, 2% index/98% data in internal nodes
-·-·- MD/OD-RT-1, utilizing one dimension of index
—— MD/OD-RT-2, utilizing one dimension of index
—— MD-RT, utilizing one dimension of index

Graph 5.14: page size = 1024 bytes; data relation 2.

PERCENTAGE OF NODES ALLOCATED ON MAGNETIC OR OPTICAL DISK, AS A FUNCTION OF THE NO. OF RECORDS

X Axis = number of records
Y Axis = percentage of nodes allocated on magnetic or optical disk



- - MD/OD-RT-1: percent of nodes on magnetic disk
····· MD/OD-RT-1: percent of nodes on optical disk
—— MD/OD-RT-2: percent of nodes on magnetic disk under root 1
-·-·- MD/OD-RT-2: percent of nodes on magnetic disk under root 2
—— MD/OD-RT-2: percent of nodes on optical disk (under root 2)

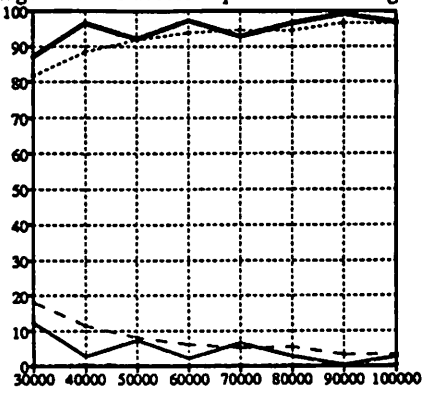
Graph 5.15: page size = 1024 bytes; data relation 1.

Graph 5.18 shows the average number of pages accessed per insert. This graph along with Graph 5.9 of the varying page size experiments clearly shows that the 2% index record per non-leaf node version of the Allocation Tree had very poor insert performance. The notable feature of these graphs is that the Allocation Trees with 2% index records in the non-leaf nodes required a larger number of accesses per insert, especially with page sizes less than 8 Kb. The results of the search performance and insert performance experiments together point out the trade-off of the percentage of index records in the non-leaf nodes of Allocation Trees. A lower percentage provides better search performance but has more costly inserts, whereas a higher percentage provides better insert performance but worse search performance. As previously mentioned, the *number* of non-leaf node entries used for index records should be at least 2 to provide reasonable search performance, for otherwise the trees degenerate into lists. Therefore, the *percentage* of non-leaf node entries used for index records should be maintained above a certain minimum to guarantee good search performance, where that percentage would depend on the page size. From the experiments, it appears that a good balance can be achieved by having 10% to 25% of non-leaf node space used for index records.

Graph 5.19 shows the sizes of all of the indexes as a function of the number of records. Both the R-Trees and Allocation Trees showed a linear growth rate, and the R-Trees were larger by approximately 25%. However, the R-Trees provided substantial search performance improvement, as was made evident by Graphs 5.11 through 5.14 for several data relation sizes.

PERCENTAGE OF NODES ACCESSED PER SEARCH ON MAGNETIC OR OPTICAL DISK FOR QUERY 1, AS A FUNCTION OF THE NO. OF RECORDS

X Axis = number of records
Y Axis = percentage of nodes accessed per search on magnetic or optical disk

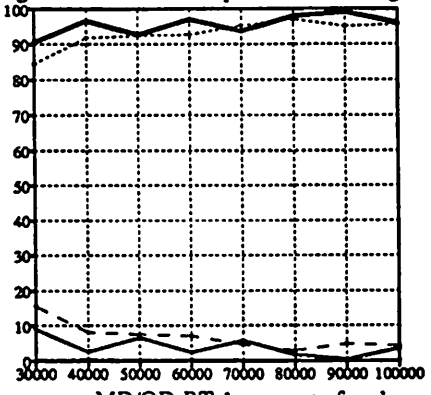


- - MD/OD-RT-1: percent of nodes on magnetic disk
····· MD/OD-RT-1: percent of nodes on optical disk
—— MD/OD-RT-2: percent of nodes on magnetic disk under root 1
-·-·- MD/OD-RT-2: percent of nodes on magnetic disk under root 2
—— MD/OD-RT-2: percent of nodes on optical disk (under root 2)

Graph 5.16: page size = 1024 bytes; data relation 1.

PERCENTAGE OF NODES ACCESSED PER SEARCH ON MAGNETIC OR OPTICAL DISK FOR QUERY 2, AS A FUNCTION OF THE NO. OF RECORDS

X Axis = number of records
Y Axis = percentage of nodes accessed per search on magnetic or optical disk



- - MD/OD-RT-1: percent of nodes on magnetic disk
····· MD/OD-RT-1: percent of nodes on optical disk
—— MD/OD-RT-2: percent of nodes on magnetic disk under root 1
-·-·- MD/OD-RT-2: percent of nodes on magnetic disk under root 2
—— MD/OD-RT-2: percent of nodes on optical disk (under root 2)

Graph 5.17: page size = 1024 bytes; data relation 1.

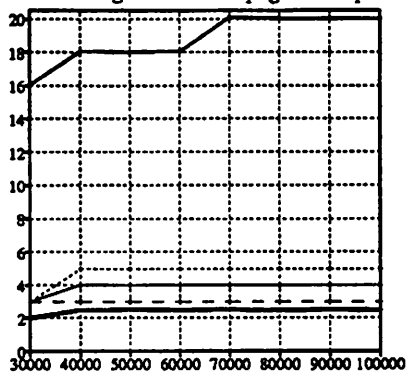
5.4. Summary and Conclusions

Two indexing structures which spanned magnetic and optical disk media were shown to have superior search performance compared to an indexing structure that was completely contained on optical disk, and these indexes had search performance equal to that of an index that was entirely contained on magnetic disk. The test database relations used in this study were constructed to reflect the properties of a historical data archive, namely, that the data in the time domain was inserted in approximate time-sorted order and the data (non-temporal) attributes were not correlated with the time attributes. Two data relations were indexed, one with a relatively long data domain and short time domain with short time ranges per record, and the other with a relatively short data domain and long time domain with long time ranges per record.

The two mixed-media indexing structures which were shown to be useful for indexing historical data relations were variations of the R-Tree index. Both variations had search performance that was identical to an R-Tree that was wholly contained on magnetic disk, and all outperformed the Allocation Trees while maintaining approximately 90% of their nodes on optical disk. The Allocation Tree, which had 100% of its nodes on optical disk, provided search performance that was inferior to that of the R-Trees. The results for the two data relations used as input and the two queries used in the search experiments were similar.

AVERAGE NO. OF PAGES READ PER INSERT USING ALLOCATION TREE AND R-TREE, AS A FUNCTION OF THE NO. OF RECORDS

X Axis = number of records
Y Axis = average number of pages read per insert

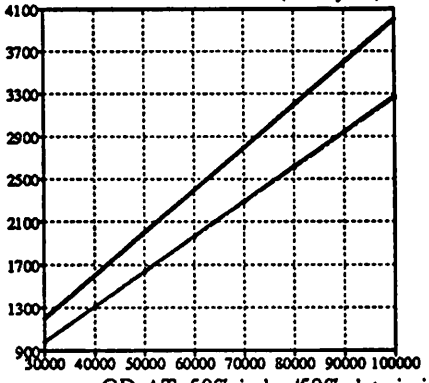


- - OD-AT, 50% index/50% data in internal nodes
····· OD-AT, 25% index/75% data in internal nodes
—— OD-AT, 2% index/98% data in internal nodes
-·-·- MD/OD-RT-1, utilizing one dimension of index
—— MD/OD-RT-2, utilizing one dimension of index
—— MD-RT, utilizing one dimension of index

Graph 5.18: page size = 1024 bytes; data relation 1.

SIZE OF ALLOCATION TREE AND R-TREE INDEXES, AS A FUNCTION OF THE NO. OF RECORDS

X Axis = number of records
Y Axis = size of index (kilobytes)



- - OD-AT, 50% index/50% data in internal nodes
····· OD-AT, 25% index/75% data in internal nodes
—— OD-AT, 2% index/98% data in internal nodes
-·-·- MD/OD-RT-1
—— MD/OD RT-2
—— MD-RT

Graph 5.19: page size = 1024 bytes; data relation 1.

While the results of this performance study are promising, future research is needed to explore the performance of mixed-media indexes over a broader collection of data types and distributions, relation sizes, insertion orders, and vacuuming frequencies and quantities other than those used in these experiments. In addition, write-once variations of indexing structures other than the R-Tree should be explored, and comparisons with alternative mixed-media approaches, such as the Write-Once B-Tree [LOME89b], should be investigated.

CHAPTER 6

CONCLUSION

The focus of this work has been the design and analysis of indexing structures and techniques for historical data, and more generally for improving the performance of a class of spatial access methods. Section 1 summarizes the results presented in Chapters 3-5. Section 2 provides a comparison of this work with other research. Finally, Section 3 discusses possibilities for future research on indexing techniques for historical and spatial data in a DBMS.

6.1. Summary

In Chapter 3, the notion of *Segment Indexes* was introduced. The idea of Segment Indexes is to combine the basic concept of the Segment Tree [BENT77] with that of a class of database indexing structures which are based on balanced, multi-way, tree-structured indexes. The primary motivation for combining these approaches was to index data that represents historical data by a set of intervals in the time domain. However, such an approach may also be extended to apply to multi-dimensional interval data, such as rectangles in two dimensions, and hyper-rectangles in d dimensions, $d > 2$. The essential idea of Segment Indexes is to store a spatial object in the highest level node that the object spans (covers) in some dimension. In order to make this idea work in traditional database indexing structures, certain modifications to those indexes were required. Those modifications included (1) allowing data to be stored in non-leaf as well as leaf nodes, (2) allowing

the page size in the index to be variable, and (3) allowing the index to be partially pre-allocated based on an estimate of the data distribution. The first modification was a departure from conventional indexes which normally store data records only in the leaf nodes. The second modification was required since without it the data records stored in the non-leaf nodes would reduce the number of node entries available for index records, and would thus diminish the fanout of those nodes. The idea of the second modification was to allow higher level non-leaf nodes to be larger than their descendants. The third modification was to construct so-called *Skeleton Indexes*, which are pre-allocated and pre-constructed indexes that are based on estimates on the amount of data to be indexed, and its distribution. By building Skeleton Indexes and then later adapting them to the actual input data, the resulting index would have better search performance characteristics as compared to a similar index that was built "from scratch". These ideas were incorporated into the R-Tree index [GUTT84], and through the result of performance experiments were shown to be sound. In particular, the Segment R-Tree provided substantially better performance as compared to R-Trees for both line segment and rectangle data.

In Chapter 4, the concept of *Lop-Sided Indexes* was presented. The motivation for Lop-Sided Indexes is to support queries that are non-uniform in the index key domain. In particular, for historical data indexes, it is likely that queries on more recent historical data will occur more frequently than queries on older historical data. However, the utility of Lop-Sided Indexes extends to any database application that will generate a non-uniform query distribution. Another motivating influence is the advent of large capacity optical disks, which have changed the assumptions about what are *typical* database relation sizes. Since historical data relations are likely to be stored on optical disks and will become quite large, their indexes must be quite large as well. For such large indexes, it is no longer true that a balanced index

would only require at most a small number of levels, such as 3 or 4. The chapter included a report of a simulation study of Lop-Sided Indexes based on the B+-Tree, using three different approaches to the problem of dynamically reorganizing the index as it evolves. These three approaches were designated (1) no-shuffle, (2) skeleton, and (3) shuffle. The first approach did no reorganization, the second constructed an initial Lop-Sided Skeleton Index based on estimates of the input data distribution and quantity, and the third performed a limited amount of index reorganization to maintain the proper degree of "lop-sidedness". The Lop-Sided B+-Tree was implemented, using each of these approaches to reorganization, and the results showed that the skeleton technique provided the best overall performance (assuming the estimates were reasonably accurate), followed by the shuffle method and then the no-shuffle method. In all cases, the performance of Lop-Sided B+-Trees relative to standard B+-Trees was shown to be as good or better, for a range of non-uniform query distributions.

In Chapter 5, the notion of *Mixed-Media Indexes* was introduced. The idea of mixed-media indexes is that historical data may be contained in a *temporally partitioned store* such that the current data and their associated indexes are maintained separately from the historical data and their associated indexes, possibly on different media. In particular, historical data and indexes may be contained in large archives on optical disk. For such a storage architecture, one approach to indexing a historical data relation is to build a *composite* or *mixed-media* indexing structure that has some component on magnetic disk and some component on optical disk. The idea was to allow the magnetic disk portion to grow until its size reaches a threshold upper limit, and at that time portions of the index are migrated, or *vacuumed*, to optical disk. Two alternative mixed-media R-Trees were compared to a magnetic disk-based R-Tree, and an optical disk-based Allocation Tree [VITT85]. The results of

these experiments showed that the mixed-media R-Trees had search performance that was as good as the magnetic-disk based R-Tree in terms of the number of pages accessed per search, and always performed better than the Allocation Tree by the same measure.

6.2. Comparison with Other Research

Research in spatial data structures for database indexing has become quite active, and has become a significant field in its own right. This thesis contributes to this field in several respects.

In Chapter 3, the idea of Segment Indexes showed how a concept from the Segment Tree binary tree data structure could be generalized for a multi-way tree, and could be adapted in an area-based spatial access method. This technique is useful for both line segment and rectangle data. In addition, the notion of adaptable Skeleton Trees is a method for pre-partitioning the data space so that indexes that use the technique of overlapping minimally bounding rectangles will have better multi-dimensional resolution. This technique provides much of the benefit of the static packing scheme of [ROUS85] or the periodic reorganization scheme of "forced reinserts" of [BECK90], and is more of a dynamic approach since it adapts to the actual input data.

In Chapter 4, the Lop-Sided Indexes were shown to be a successful extension of multi-way search trees for non-uniform query distributions. To date, no proposals have specifically addressed the problem of indexing large databases for such non-uniform query distributions, largely because the uniform query distribution assumption is widely accepted, and until recently it was believed that indexes based on balanced multi-way trees would almost never exceed more than some small number of levels, such as 3 or 4. The advent of optical disks that make it possible to store large

historical data archives overturns both of these assumptions simultaneously. Ironically, some proposals [ELMA90, LOME89b] did mention that query distributions on historical data would likely be non-uniform, but none have considered relaxing the balance criterion for tree structured indexes.

In Chapter 5, the mixed-media index approach is similar in some respects to the Time-Split B-Tree (TSBT) [LOME89b, LOME90], in that both the approach proposed in this thesis and the TSBT assume that the historical index spans magnetic and optical disk. The difference is that the TSBT migrates index nodes to optical disk one-at-a-time when a node splits, whereas the approach of Chapter 5 was to have several indexing nodes transferred simultaneously to optical disk by an asynchronous daemon process. In other respects the TSBT is fundamentally different from the mixed-media R-Tree indexes presented in Chapter 5. For example, the TSBT indexes time intervals by keeping the start time timestamp in the index records, and therefore a large number of redundant index records will probably be required in many cases. Also, since the TSBT is a balanced tree, it is likely to have several levels as the size of the historical archive becomes large. This problem is compounded by the redundant data contained in the index. The ideas developed in Chapter 5 would successfully extend to a TSBT as well as to the Time Index [ELMA90], assuming a non-uniform query distribution.

6.3. Directions for Future Research

The idea of Segment Indexes was shown to be applicable to R-Trees. While the approach of Segment Indexes generalizes to a class of multi-way, tree-structured spatial access methods, it remains to apply this approach to other indexing structures. For example, it would be interesting to apply the Segment Index approach to other variants of the R-Tree, such as the R*-Tree [BECK90] or the R-File [HUTF90].

The concept of Lop-Sided Indexes was successfully combined with the B+-Tree, to produce Lop-Sided B+-Trees. This approach may apply to any tree-structured multi-way index, particularly one in which the index records may be totally ordered in the index key domain. Such is not the case in certain area-based spatial access methods, such as the R-Tree. Further research should be devoted to determining what other indexes would benefit from the Lop-Sided Index approach.

Mixed-media indexes were applied to R-Trees, and may certainly be applied to other area-based spatial access methods that index historical data. In those experiments, the performance of the mixed-media R-Trees was essentially equal to that of the R-Tree maintained entirely on magnetic disk, in terms of the average number of nodes accessed per search. Further research is required to determine under what conditions these three R-Tree variations differ in search performance. In addition, the approach of *vacuuming* index nodes periodically from magnetic to optical disk may be applied to other spatial access methods that are used in a temporally partitioned storage architecture, and other *vacuuming algorithms* may be devised.

Supporting spatial and historical data in database management systems adds new functionality that is either desired or required by many database applications, and in addition it may also benefit the database system itself. In particular, subsystems that deal with concurrency control, crash recovery, rules, and synchronization of distributed databases may be able to use the spatial or historical attributes of data or meta-data to their advantage. Further research is required in these areas.

The results of the research contained in this thesis indicate that new access methods for spatial and historical data are worth implementing in a database system. More generally, existing spatial access methods for multi-dimensional interval data can be improved upon. All of the indexing techniques presented in this thesis

are straightforward and are not difficult to implement.

Database management systems that provide support for spatial and historical data expand the possible domain of database applications. This thesis has contributed to some of the problems posed to designers and implementors of such systems. In the future, database system prototypes may be built that utilize the techniques presented in this thesis, and applications may be developed that access large spatial databases and historical data archives. In such a system, the techniques presented here may be refined further, and performance comparisons can be made based on a variety of queries over large quantities of actual spatial and historical data.

BIBLIOGRAPHY

[AHN86] Ahn, Ilsoo. "Performance Modeling and Access Methods for Temporal Database Management Systems", PhD Thesis, Department of Computer Science, University of North Carolina, August 1986.

[AHN88] Ahn, I., and Snodgrass, R. "Partitioned Storage for Temporal Databases", *Information Systems*, Vol. 13, No. 4, 1988.

[ARAG89] Aragon, C., and Seidel, R. "Randomized Search Trees", *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science*, October 1989.

[BAYE72] Bayer, R., and McCreight, E. "Organization and Maintenance of Large Ordered Indexes", *Acta Informatica*, Vol. 1, No. 3, 1972.

[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. "Algorithms for Klee's Rectangle Problems", Computer Science Department, Carnegie-Mellon University, Pittsburgh, 1977.

[BENT85] Bent, S., Sleator, D., Tarjan, T. "Biased Search Trees", *SIAM Journal on Computing*, Vol. 14, 1985.

[BLAN90] Blanken, H., Ijbema, A., Meek, P., van den Akker, B. "The Generalized Grid File: Description and Performance Aspects", *Proceedings of the IEEE Sixth International Conference on Data Engineering*, February

1990.

[COME79] Comer, D. "The Ubiquitous B-Tree", *Computing Surveys*, Vol. 11, No. 2, June 1979.

[DADA84] Dadam, P., Lum, V., Werner, H. "Integration of Time Versions into a Relational Database System", *Proceedings of the Tenth International Conference on Very Large Data Bases*, August 1984.

[EAST86] Easton, M. "Key-Sequence Data Sets on Indelible Storage", *IBM Journal of Research* and *Development*, Vol. 30, No. 3, May 1986.

[EDEL80] Edelsbrunner, H. "Dynamic Rectangle Intersection Searching", *Institute for Information Processing*, Report 47, Technical University of Graz, Austria, 1980.

[EDEL82] Edelsbrunner, H. "Intersection Problems in Computational Geometry", PhD Thesis, Report 93, Technical University of Graz, Austria, 1982.

[ELMA90] Elmasri, R., Wuu, G., Kim, Y. "The Time Index: An Access Structure for Temporal Data", *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, August, 1990.

[FREE87] Freeston, M. "The BANG File: A New Kind of Grid File", *Proceedings of the* 1987 *ACM SIGMOD International Conference on Management of Data*, May 1987.

[FROS90] Frosh, R. "A Method of Accessing Large Spatial Databases", *Proceedings of the GIS* 1990 *Symposium*, March 1990.

[GREE89] Greene, D. "An Implementation and Performance Analysis of Spatial Data Access Methods", *Proceedings of the IEEE Fifth International Conference on Data Engineering*, February 1989.

[GUNA90] Gunadhi, S., Segev, A. "Efficient Indexing Methods for Temporal Relations", Submitted to *IEEE Knowledge* and *Data Engineering*, 1990.

[GUNT89] Gunther, O. "The Design of the Cell Tree: An Object-Oriented Index Structure for Geometric Databases", *Proceedings of the IEEE Fifth International Conference on Data Engineering*, February 1989.

[GUTT84] Guttman, A. "R-Trees: A Dynamic Index Structure for Spatial Searching", *Proceedings of the* 1984 *ACM SIGMOD International Conference on Management of Data*, June 1984.

[HANS90] Hanson, E., Chaabouni, M., Kim, C., Wang, Y. "A Predicate Matching Algorithm for Database Rule Systems", *Proceedings of the* 1990 *ACM SIGMOD International Conference on Management of Data*, June 1990.

[HENR89] Henrich, A., Six, H., Widmayer, P. "The LSD Tree: Spatial Access to Multidimensional Point and Non-Point Objects", *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, August, 1989.

[HERO80] Herot, C. "Spatial Management of Data", *ACM Transactions on Database Systems*, Vol. 5, No. 4, December 1980.

[HINR83] Hinrichs, K., Nievergelt, J. "The Grid File: A Data Structure Designed to Support Proximity Queries on Spatial Objects", *Proceedings of the WG'83 (International Workshop on Graph-Theoretic Concepts in Computer Science)*, Trauner Verlag, Linz, Austria, 1983.

[HOUG62] Hough, P. "Method and Means for Recognizing Complex Patterns", *U.S. Patent No.* 3069654, 1962.

[HP89] Hewlett-Packard Co. "Rewritable Optical Disk Library System Technical Reference Manual", *HP Part No.* 5959-3540, Edition 2, December 1989.

[HUTF90] Hutflesz, A., Six, H., Widmayer, P. "The R-File: An Efficient Access Structure for Proximity Queries", *Proceedings of the IEEE Sixth International Conference on Data Engineering*, February 1990.

[JAGA90a] Jagadish, H. "Spatial Search with Polyhedra", *Proceedings of the IEEE Sixth International Conference on Data Engineering*, February 1990.

[JAGA90b] Jagadish, H. "On Indexing Line Segments", *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, August, 1990.

[KNUT73] Knuth, D. *The Art of Computer Programming*, Vol. 3, *Sorting* and *Searching*, Addison-Wesley, Reading, MA, 1973.

[KOLO89] Kolovson, C., Stonebraker, M. "Indexing Techniques for Historical Databases", *Proceedings of the IEEE Fifth International Conference on Data Engineering*, February 1989.

[KRIE84] Kriegel, H. "Performance Comparison of Index Structures for Multikey Retrieval", *Proceedings of the* 1984 *ACM SIGMOD International Conference on Management of Data*, 1984.

[LMSI87] Laser Magnetic Storage International Co., "LaserDrive 1200 Intelligent Digital Optical Disk Drive User Manual", LMSI Co., Colorado Springs, CO, 1987.

[LOME89a] Lomet, D., Salzberg, B. "Access Methods for Multiversion Data", *Proceedings of the* 1989 *ACM SIGMOD International Conference on Management of Data*, June 1989.

[LOME89b] Lomet, D., Salzberg, B. "A Robust Multi-Attribute Search Structure", *Proceedings of the IEEE Fifth International Conference on Data Engineering*, February 1989.

[LOME90] Lomet, D., Salzberg, B. "The Performance of a Multiversion Access Method", *Proceedings of the* 1990 *ACM SIGMOD International Conference on Management of Data*, May 1990.

[LUM84] Lum, V., Dadam, P., Erbe, R., Guenauer, J., Pistor, P., Walch, G., Werner, H., Woodfill, J. "Designing DBMS Support for the Time Dimension", *Proceedings of the* 1984 *ACM SIGMOD International Conference on Management of Data*, June 1984.

[LUM85] Lum, V., Dadam, P., Erbe, R., Guenauer, J., Pistor, P., Walch, G., Werner, H., Woodfill, J. "Design of an Integrated DBMS to Support Advanced Applications", *Proceedings of the International Conference on Foundations of Data Organization*, May 1985.

[MEHL84] Mehlhorn, K. *Sorting* and *Searching*. Springer, 1984.

[MORT66] Morton, G. "A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing", IBM Ltd., Ottawa, Canada, 1966.

[McCR85] McCreight, E. "Priority Search Trees", *SIAM Journal on Computing*, Vol. 14, No. 2, May 1985.

[NIEV73] Nievergelt, J., Reingold, E. "Binary Search Trees of Bounded Balance", *SIAM Journal on Computing*, Vol. 2, 1973.

[NIEV81] Nievergelt, J., Hinterberger, H., Sevcik, K. "The Grid File: An Adaptable, Symmetric Multikey File Structure", *Trends in Information Processing Systems, Proceedings of the Third ECI Conference*, Lecture Notes in Computer Science 123, Springer Verlag, 1981.

[OHSA83] Ohsawa, Y., Sakauchi, M. "BD-Tree: A New N-Dimensional Data Structure with Efficient Dynamic Characteristics", *Proceedings of the Ninth World Computer Congress, IFIP* 83, 1983.

[OHSA90] Ohsawa, Y., Sakauchi, M. "A New Tree Type Data Structure with Homogeneous Nodes Suitable for a Very Large Spatial Database", *Proceedings of the IEEE Sixth International Conference on Data Engineering*, February 1990.

[OOI87] Ooi, B., McDonell, K., Sacks-Davis, R. "Spatial K-D Tree: An Indexing Mechanism for Spatial Database", *Proceedings of the Eleventh International Computer Software* and *Applications Conference* (COMPSAC), October 1987.

[OOI89] Ooi, B., Sacks-Davis, R., McDonell, K. "Extending a DBMS for Geographic Applications", *Proceedings of the IEEE Fifth International Conference on Data Engineering*, February 1989.

[OREN84] Orenstein, J., Merrett, T. "A Class of Data Structures for Associative Searching", *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, April 1984.

[PREP85] Preparata, F., Shamos, M. *Computational Geometry, An Introduction*, Springer-Verlag, New York, 1985.

[RATH84] Rathmann, R. "Dynamic Data Structures on Optical Disks", *Proceedings IEEE Computer Data Engineering Conference*, April 1984.

[ROBI81] Robinson, J. "The K-D-B Tree: A Search Structure for Large Multidimensional Dynamic Indexes", *Proceedings of the* 1981 *ACM SIGMOD International Conference on Management of Data*, June 1981.

[ROTE87] Rotem, D., Segev, A. "Physical Organization of Temporal Data", *Proceedings of the IEEE Third International Conference on Data Engineering*, February 1987.

[ROUS85] Roussopoulos, N., Leifker, D. "Direct Spatial Search on Pictorial Databases Using Packed R-Trees", *Proceedings of the* 1985 *ACM SIGMOD International Conference on Management of Data*, June 1985.

[SALZ89] Salzberg, B., Lomet, D. "Robust Index Structures for Large Spatial Databases", College of Computer Science, Northeastern University, Technical Report NU-CCS-89-16, 1989.

[SAME88] Samet, H. "Hierarchical Representations of Collections of Small Rectangles", *ACM Computing Surveys*, Vol. 20, No. 4, December 1988.

[SAME89a] Samet, H. *The Design* and *Analysis of Spatial Data Structures*, Addison-Wesley, Reading, MA, 1989.

[SAME89b] Samet, H. *Applications of Spatial Data Structures: Computer Graphics, Image Processing*, and *GIS*, Addison-Wesley, Reading, MA, 1989.

[SARN86] Sarnak, N., Tarjan, R. "Planar Point Location Using Persistent Search Trees", *Communications of the ACM*, Vol. 29, No. 7, July 1986.

[SEEG88] Seeger, B., Kriegel, H. "Techniques for Design and Implementation of Efficient Spatial Access Methods", *Proceedings of the Fourteenth International Conference on Very Large Data Bases*, August, 1988.

[SEEG90] Seeger, B., Kriegel, H. "The Buddy-Tree: An Efficient and Robust Access Method for Spatial Data Base Systems", *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, August, 1990.

[SEGE87] Segev, A., Shoshani, A. "Logical Modeling of Temporal Data", *Proceedings of the* 1987 *ACM SIGMOD International Conference on Management of Data*, May 1987.

[SELL87] Sellis, T., Roussopoulos, N., Faloutsos, C. "The R+-Tree: A Dynamic Index for Multi-Dimensional Objects", Department of Computer Science, University of Maryland, Technical Report CS-TR-1795, February 1987.

[SHEN90] Sheng, J., Sheng, O., "R-Trees for Large Geographic Information Systems in a Multi-User Environment", *Proceedings of the Twenty-Third Annual IEEE Hawaii International Conference on Systems Sciences, Vol. II, Software Track*, January 1990.

[SHOS86] Shoshani, A., Kawagoe, K. "Temporal Data Management", *Proceedings of the Twelfth International Conference on Very Large Data Bases*, August, 1986.

[SIX88] Six, H., Widmayer, P. "Spatial Searching in Geometric Databases", *Proceedings of the IEEE Fourth International Conference on Data Engineering*, February 1988.

[SLEA85] Sleator, D., Tarjan, T., "Self-Adjusting Binary Search Trees", *Journal of the ACM*, Vol. 32, 1985.

[SNOD85] Snodgrass, R., Ahn, I. "A Taxonomy of Time in Databases", *Proceedings of the* 1985 *ACM SIGMOD International Conference on Management of Data*, June 1985.

[SNOD86] Snodgrass, R., "Research Concerning Time in Databases: Project Summaries", ACM SIGMOD Record, Vol. 15, No. 4, 1986.

[STON87] Stonebraker, M. "The Design of the POSTGRES Storage System", *Proceedings of the Thirteenth International Conference on Very Large Data Bases*, September, 1987.

[STON90] Stonebraker, M., Jhingran, A., Goh, J., Potamianos, S. "On Rules, Procedures, Caching and Views in Data Base Systems", *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, June 1990.

[TAMM81] Tamminen, M. "The EXCELL Method for Efficient Geometric Access to Data", *Acta Polytechnica Scandinavica*, Mathematics and Computer Science Series No. 34, Helsinki, Finland, 1981.

[VITT85] Vitter, J. "An Efficient I/O Interface for Optical Disks", *ACM Transactions on Database Systems*, Vol. 10, No. 2, June 1985.

[WHAN85] Whang, K., Krishnamurthy, R. "Multilevel Grid Files", IBM Thomas J. Watson Research Center, Research Report RC 11516 (#51719), November 1985.