

CONCURRENT ACCESS TO SPATIAL DATA

by

Vincent T.Y. Ng

Master of Mathematics University of Waterloo, 1986

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
in the School
of
Computing Science

© Vincent T.Y. Ng 1994
SIMON FRASER UNIVERSITY
August 1994

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

APPROVAL

Name: Vincent T.Y. Ng
Degree: Doctor of Philosophy
Title of thesis: Concurrent Access to Spatial Data

Examining Committee: Dr. J.G. Peters
Chair

Dr. T. Kameda, Senior Supervisor

Dr. W.-S. Luk

Dr. M.S. Atkins

Dr. B.K. Bhattacharya

Dr. M. Yamashita, External Examiner

Date Approved:

August 23, 1994

SIMON FRASER UNIVERSITY


PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

Concurrent Access to Spatial Data.

Author:



(signature)

Vincent Ng

(name)

August 17, 1994

(date)

Abstract

Spatial data consist of points, lines, rectangles, polygons, and volumes, etc. The structure by means of which to organize and access such data is important to the performance of database systems which support applications in computer vision, computer-aided design, solid modeling, geographic information system and computational geometry, etc. In the past, little work has been done in developing concurrency control algorithms to access spatial data.

The thesis investigates concurrent operations on spatial index structures. We build our work on existing spatial data structures and concurrency control algorithms. Some of the concurrency control algorithms we designed has been implemented to demonstrate their effectiveness.

We first study known index structures for point data, **B⁺-tree**, **R-tree**, and **K-D-B tree**. To overcome some drawbacks of the existing index structures, we propose a new index structure, called **quad-B tree**, which combines the advantages of the B-tree and the quadtree. We control concurrent access to these index structures, using the lock-coupling or link technique.

We then study two index structures for rectangular data, **R-tree** and **quad-R tree**. We discuss different approaches to supporting concurrent operations on an R-tree, namely, the *simple*, *lock-modify*, *lock-coupling*, *give-up* and the *link* approaches. We compare the search performance of the first three approaches based on their implementations. The link technique is adopted for the R-tree to support recovery after system failures. Finally, we use the quad-R tree to solve the problem of ordering amongst spatial objects.

Acknowledgements

Many individuals have helped to make it possible for me to successfully complete this thesis. Although it is not possible to name them all, I give them my thanks. There are some who merit special recognition and I would like to take this opportunity to acknowledge their contribution.

First, I especially would like to thank my supervisor, Dr. Tiko Kameda. Without his care and direction, I am certain that this thesis would not have been completed. I am extremely fortunate to have been his student.

I am also grateful to Dr. Wo-Shun Luk, Dr. Stella Atkins and Dr. Binay Bhattacharya, not only for their patience and commenting on this thesis, but also for their constant advice and encouragement. I also thank Dr. Masafumi Yamashita, the external examiner of this thesis, for his valuable comments and suggestions.

Many thanks go also to all my colleagues in the British Columbia Cancer Agency. I am indebted to Dr. Pierre Band, Dr. Allen Eaves and Dr. Andrew Coldman for their encouragement and support throughout my study.

Finally, this work is dedicated to my wife, Louisa, for her eternal patience, support and assistance, and most importantly, her love.

Contents

Abstract	iii
Acknowledgements	iv
1 Introduction	1
1.1 Spatial Data	1
1.2 Objectives of the Thesis	2
1.3 Overview of the Thesis	3
2 Review of the Literature	5
2.1 Point Data	5
2.2 Rectangular Data	8
2.3 Concurrent Search Algorithms	11
2.3.1 Lock-Coupling	11
2.3.2 Link Technique	11
2.3.3 Give-Up Technique	11
2.4 Related Work	12
3 A GIS Database Framework	14
3.1 Operations Studied	15
3.1.1 Point Data	16
3.1.2 Rectangular Data	16
3.2 Locking Protocol	17
3.3 Definitions	17

4	Point Data	19
4.1	B ⁺ -Tree	19
4.1.1	Point Transformation	19
4.1.2	Search	23
4.1.3	Insert	25
4.1.4	Delete	27
4.1.5	Correctness of the Operations	29
4.2	R-Tree	33
4.2.1	Search	34
4.2.2	Insert	35
4.2.3	Delete	39
4.2.4	Correctness of the Operations	43
4.3	K-D-B Tree	45
4.3.1	Search	47
4.3.2	Insert	48
4.3.3	Delete	52
4.3.4	Correctness of the Operations	56
4.4	Quad-B Tree	57
4.4.1	Lock-Coupling	61
4.4.2	Link Extension	69
4.4.3	Correctness of the Operations	75
5	Rectangular Data	79
5.1	R-Simple Tree	79
5.1.1	Search	80
5.1.2	Insert	81
5.1.3	Delete	82
5.1.4	Correctness of the Operations	82
5.2	R-Lock Tree	82
5.2.1	Search	84
5.2.2	Insert	85
5.2.3	Delete	86
5.2.4	Maintain	87

5.2.5	Correctness of the Operations	89
5.3	R-couple Tree	90
5.3.1	Search	90
5.3.2	Insert	91
5.3.3	Delete	91
5.3.4	Correctness of the Operations	92
5.4	R-opt Tree	92
5.4.1	Search	94
5.4.2	Insert	99
5.4.3	Delete	104
5.4.4	Maintain	107
5.4.5	Correctness of the Operations	110
5.5	R-Link Tree	112
5.5.1	Supporting Procedures	116
5.5.2	Atomic Actions	118
5.5.3	Search	123
5.5.4	Insert	124
5.5.5	Delete	129
5.5.6	Compress	131
5.5.7	Serializability and Multiple Searches	137
5.6	Quad-R Tree	138
5.6.1	Search	140
5.6.2	Insert	145
5.6.3	Delete	148
5.6.4	The Secondary Trees	150
5.6.5	Correctness of the Operations	158
6	Comparisons and Implementations	163
6.1	Locking Protocols	164
6.2	Storage Utilization	168
6.3	Implementations	169
6.3.1	Point Data	170
6.3.2	Rectangular Data	173

7 Conclusion	177
7.1 Summary	177
7.1.1 Point Data	177
7.1.2 Rectangular Data	178
7.2 Future Work	179
7.2.1 Concurrency Control	180
7.2.2 Recovery	180
7.2.3 Other Spatial Data	181
Bibliography	182

List of Tables

List of Figures

2.1	Point data structures	6
2.2	A Grid file with different cell sizes.	7
2.3	A point quadtree of 4 levels.	7
2.4	A K-D-B tree	8
2.5	A segment tree showing the y-intervals of rectangles <i>A</i> and <i>B</i>	9
2.6	Point representation.	10
3.1	Lock compatibility matrix.	16
4.1	Z-ordering mapping.	20
4.2	A B ⁺ -tree.	20
4.3	Bit interleaving with Gray code mapping.	21
4.4	A p-G tree (<i>b</i> = 2).	22
4.5	Decomposition of a query range.	22
4.6	Search operation using the lock-coupling method.	24
4.7	Multiple sub-searches.	30
4.8	An R-tree.	34
4.9	Insert operation using the lock-coupling method.	37
4.10	Recursive splitting.	47
4.11	Joinable sets of rectangles.	48
4.12	Forced splits.	51
4.13	The shaded regions contain one or more point objects.	58
4.14	A Quad-B tree.	59
4.15	Scaler values for the terminal quadrants.	61
4.16	A Quad-B tree with link extension.	70

4.17	Lock compatibility matrix for a quad-B tree with link technique.	70
4.18	Using the link pointer to search.	71
4.19	Two steps of a split.	74
5.1	Search operations using the modify-lock method.	84
5.2	Node changes and their effects to operations>(* - Changes which will affect subsequent operations).	94
5.3	A node in the R-opt tree.	94
5.4	Search operation in an R-opt tree.	95
5.5	Log Table.	96
5.6	Phase one of inserting an object.	100
5.7	Re-organize the tree at node L	108
5.8	A node of an R-link tree.	113
5.9	Structural changes resulting from an operation	114
5.10	Using the link pointer to search.	123
5.11	Collapsing nodes in a condense operation.	132
5.12	Local reorganization.	136
5.13	Overlapping search times.	138
5.14	Five types of rectangles within a quadrant.	139
5.15	A search window divided into 4 query ranges.	141
5.16	Using the link pointer to search.	141
5.17	Performing a search.	142
5.18	Lock compatibility matrix for a quad-R tree.	151
5.19	Clusters of rectangles which separated nicely.	152
5.20	Rectangles that are hard to organize.	152
5.21	Query region for search interval $[s_l, s_r]$	153
5.22	Three types of line intervals.	153
5.23	Linearization of the \mathcal{D} space.	154
5.24	Restructing of a subtree in a quad-R tree.	161
6.1	Index structures studied in Chapter 4 and 5.	164
6.2	Summary characteristics of locking protocols in the index structures (M is the maximum number of entries within a node and n is the number of all the leaf nodes).	166

6.3	Summary characteristics of concurrent operations which can be active in the index structures.	167
6.4	Summary characteristics of storage utilization in the index structures (M is the maximum number and m is the minimum of entries within a node). . . .	169
6.5	Implementation Structure	170
6.6	Search performance with no insert operation. (Average search time in μ seconds versus the number of nodes).	171
6.7	Search performance with 50 insert operations. (Average search time in μ seconds versus the number of nodes).	172
6.8	Storage utilization of different index structures. (Average node size versus the number of nodes).	173
6.9	An R-tree implementation.	174
6.10	Search performance with 10% insert operations (Avg. search time in μ seconds versus no. of nodes).	174
6.11	Search performance with 50% insert operations. (Avg. search time in μ seconds versus no. of nodes).	175
6.12	Search performance with an R-tree containing 1000 objects (Avg. search time in μ seconds versus no. of nodes).	176

Chapter 1

Introduction

1.1 Spatial Data

During the last decade, the research and development in databases have moved into non-traditional applications, such as CAD, VLSI (very large-scale integration), multi-media, vision, computer-aided design, solid modeling, geographic information system (**GIS**) and computational geometry. In these non-standard applications, large sets of spatial objects with geometric attributes have to be stored. There are distinct differences between these applications and conventional databases that are developed for inventory, accounting and other commercial systems.

Spatial data consist of points, lines, rectangles, polygons, and volumes, etc. Operations supported for geometric manipulations (e.g., rotation and translation of objects) and spatial queries (e.g., intersection and containment) are much harder to implement than the operations in conventional database systems.

A typical GIS contains a database storing a collection of data objects of multi-dimensions. These spatial objects *intersect* with, are *neighbor* of, or *enclose* other objects. Typical database queries refer to a spatially clustered subset of the set of objects. For example, a query can be: "Find all the towns within the given ranges of longitudes and latitudes." Queries of this type are called *window queries*. A window query selects spatial objects which overlap with the given query window. An auxiliary index structure for accessing spatial data is particularly important to the performance of database systems. One of the major difficulties in such an index structure is to organize spatial objects linearly. Spatial index structures, which pre-process and store spatial relationships among spatial objects for

future queries, do not always have good search performance.

In order to efficiently find spatial objects by a line parallel to one of the coordinate axes, spatial indices which utilize spatial information have been developed. Even though there has been a lot of work on the index structures for spatial data, there is little work on concurrency control of accesses to them. While accessing an index structure, the whole index structure is commonly locked. This limits access to the index information to one process at a time.

1.2 Objectives of the Thesis

In order to allow concurrent operations, this thesis develops algorithms and protocols based on existing spatial data structures and concurrency control protocols. Some of them will be implemented to demonstrate their effectiveness. To study the problem in as simple a framework as possible, our model is limited to 2-dimensional objects, but most results can be extended to higher dimensions along the same principles.

We first consider index structures for 2-dimensional point data. The first point index structure we study is the **B⁺-tree**. The second point index structure is the **R-tree** which allows bounding rectangles of spatial objects to overlap. An R-tree is a height-balanced tree similar to a B⁺-tree, with the records in its leaf nodes containing references to data objects. Bounding rectangles of entries within the same node are allowed to overlap. The **K-D-B tree** is the third point index structure that we will discuss. A K-D-B tree partitions the search space in a manner similar to a **kd-tree**. In a kd-tree, a given a search window is partitioned into two smaller windows based on the comparison with some element of one of the domains of the coordinates of data points. For the above data structures, we will adopt the *lock-coupling* approach to support concurrent access. In order to gain a better insight, we will implement some of the algorithms. We will develop a new point index structure, called the **Quad-B tree**. It is a B⁺-tree structure which stored non-empty quadrants, which are linearly ordered by some encoding scheme. We will use the *link* approach and the *lock-coupling* approach in accessing this data structure.

We study two index data structures for rectangular data. The first data structure is the *R-tree*. We use five different approaches to support concurrent operations on an R-tree. They are the *simple* approach, *lock-modify* approach, *lock-coupling* approach, *give-up* approach and the *link* approach. We shall compare these approaches with respect to

implementation difficulties and locking requirements. We will implement the first three approaches to evaluate their search performance. We will also adopt the *link* approach to the R-tree to support recovery after system failures. We will then introduce the **Quad-R tree** which attempts to solve the problem of ordering spatial objects.

1.3 Overview of the Thesis

In Chapter 2, we review the literature on index structures for point and rectangular data. We start with the discussion of the most popular data structures to index point objects. We then move on to present the three major classes of representations for rectangular objects. We will also discuss the three different approaches to extend a data structure for concurrent operations. Finally, we comment on the **Parallel-R tree** and the **R^{link}-tree**, which have been proposed recently.

In Chapter 3, we will present a GIS database framework on which our study is based. We also describe the types of operations for the index structures we studied. Before the end of the chapter, we discuss the different types of locks used on the nodes of index structures and introduce some notations to simplify the presentations of algorithms in later chapters.

In Chapter 4, we study concurrent operations on different point data structures. We will show how to use the *lock-coupling* technique together with the *Gray code* mapping to index point data by the **B⁺-tree** structure. Then, we adopt the same concurrency approach to the **R-tree**. To reduce the number of multiple search paths, we propose the **K-D-B tree** together with *forced splitting* to avoid recursive splits. Finally, we develop a new data structure, called **Quad-B tree**, to support concurrent access by means of the *link* technique.

In Chapter 5, we continue our study of spatial data but change the focus to the rectangular data. We extend the **R-tree** with different concurrency approaches, i.e., to make it amenable to *simple*, *lock-modify*, *lock-coupling*, *give-up* and *link* approaches. We also discuss the advantages of and problems with the different approaches. The **Quad-R tree** is the final rectangular index structure that we discuss. This tree utilizes a transformation to represent non-empty quadtrees by a B⁺-tree which enables us to adopt the *link* approach for concurrent operations.

In Chapter 6, we discuss and compare the performance of the different spatial data structures in the previous two chapters. For the point index structures, we implement the algorithms for the *B⁺-tree*, the *R-tree* and the *K-D-B tree* in a distributed programming

language called **SR**. For the rectangular index structures, we implement the algorithms for the R-tree with the *simple*, *modify-lock* and *lock-coupling* approaches. We will present the program modules and the simulation setup for the index structures. The results are preliminary but they reflect the trade-offs between different algorithms.

Finally, in Chapter 7, we summarize our contributions and suggest possible future extensions to our work.

Chapter 2

Review of the Literature

In this chapter, we briefly review different index structures for the point and rectangular spatial data. We will discuss three concurrency control approaches: the *link* technique, the *lock-coupling* approach and the *give-up* approach. These approaches are adopted to support concurrent operations. Some recent research, such as the *Parallel R-tree* and *R^{link}-tree*, will also be described.

2.1 Point Data

Multidimensional point data can be represented in a variety of ways. The representation chosen for a specific task is heavily influenced by the types of operations to be performed on the data. In [Sam82], three major categories of structures are discussed according to different search techniques used. The first category represents structures which linearize point data into a sequential list. The second category includes structures which store data according to their embedding space. Examples are the grid method, EXCELL[Tam81], and MX quadtree[AbW88]. The third category includes structures which store data according to their spatial locations, such as grid file[NHS84], dynamically quantized pyramid[Slo81], point quadtree, kd-tree[Ben75], and adaptive kd-tree [MHN84]. There are some hybrid structures such as PR quadtree and PR kd-tree[Sam82] which have features of both the second and the third categories. Figure 2.1 shows the evolution of the point index structures from top to bottom.

When point data are organized into a sequential list, in most cases, spatial information is not well utilized and the worst performance of a search operation is $O(n)$, where n is the

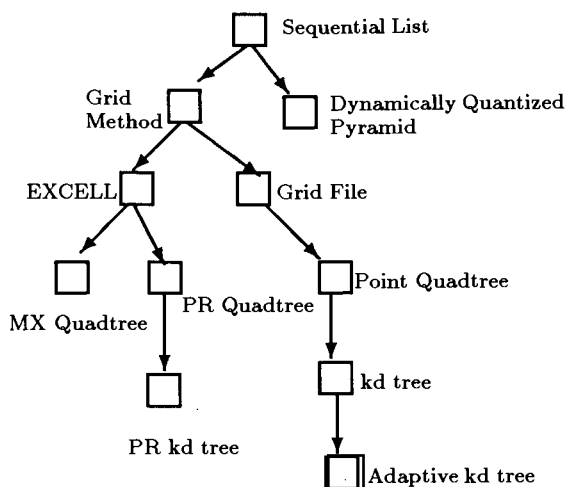


Figure 2.1: Point data structures

number of points. In general, there is no good way of ordering the data points linearly while preserving spatial proximity as much as possible. Some transformation techniques [Hin85, SeK88] have been suggested which associate data point locations with scalar values so that the resulting values can be stored in a B-tree. One of the most popular transformations is bit-interleaving [OrM84] which takes the bits of the binary representations of the coordinates alternately to obtain a scalar value (see Section 4.1). In [Fal86], Faloutos has concluded that some pre- and post-transformation using the Gray code before bit-interleaving can lead to “improved performance.”

The index structures in the second category organize the data by partitioning the data space into a fixed number of cells. The grid method divides the space evenly and the Dynamically Quantized Pyramid divides the space with varying sizes of cells. The grid method can be refined to allow varying number of cells. If the cell boundaries are fixed, it is the EXCELL structure. All of the above structures are generally represented as arrays. To improve the search performance, the EXCELL structure can be further refined to either MX quadtree or PR quadtree by allowing the partitioning of a cell into smaller sub-cells without affecting its neighbor cells. Data can now be organized as hierarchical structures. The PR kd-tree is a further refinement of the PR quadtree by allowing a cell to be decomposed into two sub-cells only and the partitioning at different levels cycle through the attributes in the data.

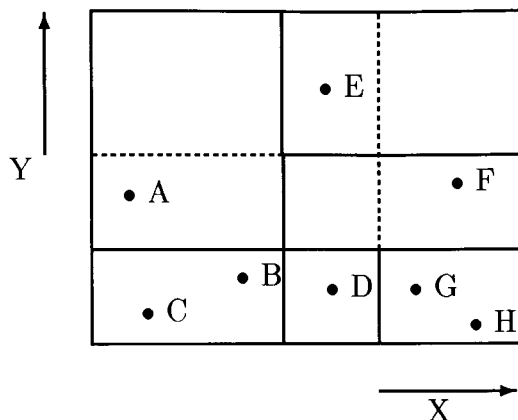


Figure 2.2: A Grid file with different cell sizes.

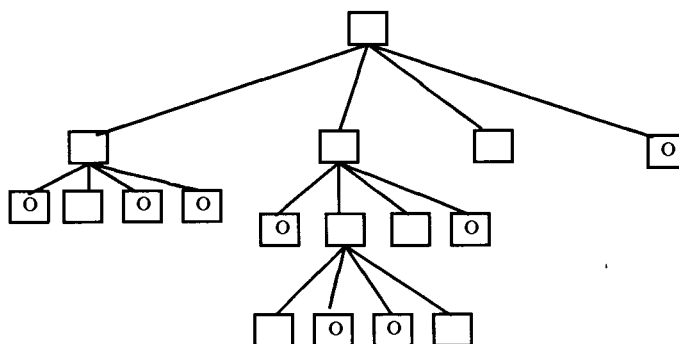


Figure 2.3: A point quadtree of 4 levels.

The grid file structure (see Figure 2.2) belongs to the third category. It is similar to the EXCELL structure except that cell boundaries are not fixed. Analogous to the MX quadtree, the point quadtree is a refinement of the grid file structure. The kd-tree is an improvement upon the point quadtree (see Figure 2.3) obtained by allowing flexible partitioning lines. The adaptive kd-tree refines the splitting of nodes in a kd-tree further by choosing a coordinate whose values from the points has the greatest spread. Thus, it is not require to cycle through the attributes as is the case with the kd-tree.

All the point data structures described above are generally limited to static organizations and it is difficult to support insertions and deletions on them, which makes incremental reorganization costly at best. Moreover, search paths for data accesses tend to be long.

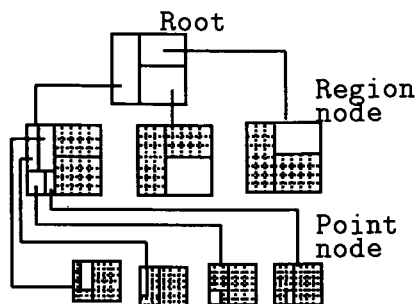


Figure 2.4: A K-D-B tree

A bucket approach has been adopted for the kd-tree which then was developed into the K-D-B tree [Rob81] (see Figure 2.4). The K-D-B tree combines the properties of a kd-tree and B-tree and it was expected to have the search efficiency of a balanced kd-tree and the storage efficiency of a B-tree. The hB-tree [LoS90] is a variant of the K-D-B tree, which attempts to improve storage utilization and allows better search and insert performance.

2.2 Rectangular Data

Rectangles are often used as approximations of shapes for which they are used as the minimum bounding boxes. They find use in GIS applications as well as VLSI design rule checking. In cartographic applications, rectangles are used to give a rough indication of the extent of an object. They can be the bounding boxes of lakes, forests, hills, etc. In VLSI, they present chip components and are used in the analysis of their proper positioning. Therefore, a geometric object of an arbitrary shape is usually characterized by its bounding box which can be used as a geometric key. In this thesis, we represent any 2-d object by its minimum bounding rectangle (**MBR**) and use it as the access index key.

In [Sam88], Samet describes three general classes of methods to represent a large collection of rectangles. The first class contains the data structures adopting the *plane-sweep* methods. Examples of the first class data structures are the segment trees [Ben77] (see Figure 2.5), interval trees [Ede80] and priority trees [McC85]. In Figure 2.5, each leaf node of the tree is an end point of the y-interval of the rectangle. Each node carries a set of labels to represent the inclusion of a y-interval and a node at a higher level includes all intervals of its descendants. The y-interval of rectangle *A* spans from the leftmost leaf node to the second

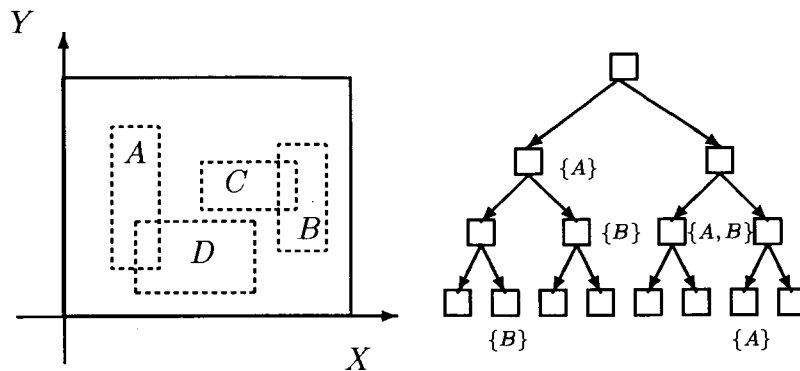


Figure 2.5: A segment tree showing the y-intervals of rectangles A and B .

rightmost leaf node. This class of methods generally use two passes. The first pass sorts the data along one dimension, and the second pass sweeps a scan line through the sorted data. Rectangles are represented by the intervals that form their boundaries. However, the methods have a poor query performance when updating of rectangles occurs frequently. The first problem with this approach is that it requires the endpoints of all rectangles to be known a priori and the points to be sorted before the sweep. A more serious problem is that the sweep pass will usually be re-executed, since there is no data structure corresponding to it.

The second class consists of the *point-based* methods which transform rectangles to higher dimensional points. Rectangles can be parameterized with location parameters or extension parameters [HiN83]. Location parameters specify the coordinates of points such as the corners, whereas extension parameters specify size, such as the width and length of a rectangle. The choice of representation affects queries differently. Proximity queries involving point and rectangular objects are easy to implement. Their answers are conic-shaped regions in the four-dimensional space. However, because the representation is not area oriented, two rectangles may be very close or overlap, yet the Euclidean distance between their representative points may be quite large. The nearness information is not captured well. In Figure 2.6, each rectangle is represented by a pair of 2-d points. For a rectangle with center at (x_c, y_c) and size $(2x_l, 2y_l)$, its point representations are (x_c, x_l) and (y_c, y_l) in the upper and lower graphs correspondingly. In the figure, rectangles A and B intersect the search window C , but it cannot be easily observed if they intersect each other except by checking their

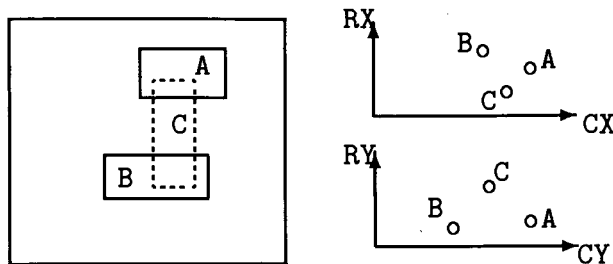


Figure 2.6: Point representation.

sizes.

The last class consists of the area-based methods, which includes MX-CIF Quadtrees[Ked83], Multiple Quadtree Blocks[AbS83, Sha86] and R-trees[Gut84]. The MX-CIF quadtree associates each rectangle (R) with the quadtree node corresponding to the smallest quadrant in which R is totally enclosed. Rectangles can be associated with both terminal and non-terminal nodes. With the total enclosure property, finding how many rectangles intersect a window may be expensive. It is because quadtree nodes that intersect the query rectangle may contain many rectangles that do not intersect the query rectangle, yet each one of them must be individually compared with the query rectangle to check for a possible intersection. The multiple quadtree block representations attempt to solve this problem. They include the expanded MX-CIF quadtree[AbS83], RR_1 quadtree and RR_2 quadtree[Sha86]. The R-tree is a hierarchical data structure derived from the B^+ -tree. Rectangles are allowed to overlap and a good storage utilization is maintained. Its difficulties are that it cannot represent adjacency, and a large number of nodes need to be examined when there are many overlaps. The R^+ -tree is an alternative to the R-tree which avoids overlap among the bounding rectangles. The cell tree in [Gun87] is similar to the R^+ -tree, the principal difference being that nonleaf nodes of the cell tree are convex polyhedra instead of bounding rectangles. Gunther implemented and investigated the performance of the cell tree [GuB91]. His results show that the cell tree required more storage space when compared with the R-tree and R^+ -tree. However, for search operations, the cell tree need a smaller number of disk accesses than the other two index structures.

2.3 Concurrent Search Algorithms

Concurrency control is the activity of preventing inconsistency of data in a database. Algorithms have been developed for search structures that produce conflict-preserving serializable histories [BHG87]. In a hierarchical data structure, there are two phases of work for the three operations that we are interested in. The first phase descends the tree from the top to the bottom in order to find the appropriate leaf node. The second phase returns a value or updates the tree as needed.

2.3.1 Lock-Coupling

Bayer and Schkolnick proposed a *lock-coupling* [BaS77] protocol that takes advantage of the B-tree structure. In this protocol, a search or update operation locks the child of a node before releasing its lock on the node itself. Thus, a process always holds at least one lock during its execution. Other variants of the protocol have since been developed [Ell80, KwW82, MiS78]. Otoo further improved lock-coupling by having three types of locks so as to allow more concurrent operations on a node [Oto90].

2.3.2 Link Technique

In [KuL80], Kung and Lehman proposed the *link* technique, which was adapted by Lehman and Yao to B-trees [LeY81]. In this approach, search, insert, and delete operations are allowed to release the lock on a node they hold before they obtain a new lock on the next node. It offers more concurrency than lock-coupling, but requires the addition of new edges or links to the structure in order to avoid anomalies. Other algorithms have been proposed that use the link technique in structures such as binary trees [MaL82], hash structures [Ell82], and B^+ -trees [LaS86].

2.3.3 Give-Up Technique

Similarly to the link technique which maintains extra information in the form of a link pointer, Shasha [ShG88] suggested the *give-up* technique which adds redundancy with a range field. Before any operation can be done on a node, his protocol checks the range of the node. If the operation's argument is not in the range, the protocol gives up and tries an ancestor of that node.

To illustrate the above three techniques better, consider a scenario in which we want to locate a book in a library. Lock-coupling is analogous to having the borrower read the library catalog, ensure that no reshelver will be working on the shelf which he wants to go to, and then go to the shelf indicated by the catalog. The link technique is analogous to the procedure in which, when a reshelver moves books on shelf m to another shelf, he leaves a note at m saying where the new shelf is, and updates the catalog. This note constitutes a link. When the borrower arrives at m and sees the note, he will then know which new shelf to go to. The give-up technique corresponds to the situation where the borrower goes to the shelf and finds a note saying that the books in question have been moved. He would then go back to the catalog and try to locate the new shelf. If the catalog has not been updated by the reshelver yet, the borrower would need to keep re-accessing the catalog until he finds out the new shelf eventually.

All of the three techniques described above have been widely discussed and implemented. Each of them possesses properties which are advantageous in different situations. Both the link technique and give-up technique require additional data structures but offer more concurrency than lock-coupling. However, both of them may suffer from livelock. When applying these techniques to spatial index structures, the link technique has the problem of merging neighborhood nodes during updates. This is because the indices in the tree are not organized in any order at all. Hence, when there is a rearrangement of link pointers at one level, it may cause a downward propagation of link pointer updates. In this thesis, we apply these three techniques to different spatial index structures.

2.4 Related Work

Concurrency control algorithms for spatial data have not been investigated extensively. In the past, most researchers in the area of spatial databases have concentrated on developing new spatial index structures and improving existing ones. There are only two projects closely related to our work. In 1993, Kamel and Faloutsos developed the **Parallel R-tree**, which tries to exploit parallelism with multiple disks to obtain better search performance[KaF92]. However, they did not report any concurrent updating algorithms for the new index structure. In their report, they suggested the use of proximities among spatial objects so as to cluster them in a small portion of an R-tree. Different portions of the tree are then put on different hard drives. When there are multiple search queries, the queries may then utilize

different drives simultaneously and this reduces the contention among them.

More interestingly, in 1994, Zhou [ZhD94] has developed the **R^{link}-tree** which uses a semantic model. He proposes an evolution tree which is a binary tree representing the evolution of a sub-R-tree from a single node during reorganization of an R-tree. The evolution tree is then incorporated into an R-tree so as to support concurrent operations. He has adopted the *link* approach to extend an R-tree with link pointers. Each node in an R-tree has the right link pointer pointing to its right sibling. The new data structure supports a high degree of concurrency and allows a simpler locking protocol. In an *R^{link}-tree*, nodes belong to different *node types*. When a node P is split into P and P' , the node P will have node type L . This is used to indicate that the tree has been changed but the parent of P has not been updated. When an operation visits P , by using P 's node type, it can determine if the right sibling need to be visited. The *R^{link}-tree* is very similar to our *R-link tree* in Chapter 5, except that each node in an R-link tree has the parent pointer (a pointer referencing its parent). The parent pointer is used similarly to the node type in the *R^{link}-tree*. In our work, the R-link is used to support concurrent operations as well as recovery after system failures.

Chapter 3

A GIS Database Framework

Spatial databases are developed to manipulate, to store, to retrieve, to analyze and to display geometric data. In general, spatial data are defined within a space by specifying their spatial coordinates. Geographic systems based on graphical properties of data have drawn much interest in recent research developments.

Geographic applications are characterized by massive volumes of data and deal with mostly 2-dimensional data. Spatial objects in these applications can be classified into two types: *spatial* and *thematic* data. Furthermore, each spatial object has two parts: geometric and topological parts. The geometric part represents the spatial object using raster or vector formats, while the topological part represents its spatial relationships with the objects (e.g., overlapping, adjacency). Geometric data in a GIS represent the spatial features of objects, such as towns and cities, highways and streets, administration areas and metropolitan regions, etc. From these examples, we see that real geometric objects can be abstracted by three types of objects: *point*, *line*, and *region*. A point data has no dimension and it refers to an object occupying a specific location in the 2-d space. A line is of one dimension and it is a connected set of straight-line segments. Each line segment is enclosed within a continuous set of line segments. A region is a two dimensional object which is represented by a set of connected line segments. Both raster and vector format can be used to represent the three types of objects. However, geometric data are represented most often in vector format because data retrieval is easier due to their spatial characteristics.

In a GIS, spatial operators such as *overlap*, *enclosure* and *adjacent* are required. These operators are very different and much more difficult to implement than the *join* and the

project operators in conventional database systems. The efficiency of these operations depends on the representation of the data and the retrieval algorithms. An indexing structure which provides fast access to spatial data is often critical in a GIS. Of the many spatial operators, we have selected the *overlap* operator, which is the most popular operator, as the main query operator in retrieving spatial information.

We study point data because of their importance in a GIS. Many GIS queries are location specific. Queries such as “Find all postal stations in the city of North Vancouver” and “Where are the fire stations in the downtown district” are some examples of queries retrieving point information. We do not study line objects in this thesis, because we believe that they can be represented as rectangular objects which are discussed in the Chapter 5. Besides queries about point objects, queries about region data from different data sources (e.g., overlaying in GIS) are popular. For example, a query like “Find all the buildings that overlap with a park” is often asked during city planning. We approximate region data by using minimum bounding rectangles (MBR), because the calculation of spatial relationships between arbitrary polygons is often hard. A region, as a polygon, may have many sides and the orientations of its line segments can vary greatly. In particular, every 2-d object except a single point can be associated with an MBR. We therefore study the concurrent accesses to axis-parallel rectangular data instead of general polygonal data.

3.1 Operations Studied

Throughout the thesis, we discuss different *insert* and *delete* algorithms to allow updating in different spatial index structures. An insertion adds a new spatial object while a deletion removes a spatial object from a GIS.

In our work, we concentrate our effort on achieving good search performance. There are two types of searches frequently used in a B⁺-tree: the *exact match search* and the *range search*. An exact match search returns data that have the same value as the key. On the other hand, the range search retrieves all data falling within a range of specified values. In an index structure, the exact match search is analogous to the *point query* for point data, while a range search is analogous to a *window search*. A window search in a GIS may be either a *window search for points* or a *window search for rectangles*.

	ρ	ω	ϵ
ρ	1	1	0
ω	1	0	0
ϵ	0	0	0

Figure 3.1: Lock compatibility matrix.

3.1.1 Point Data

There are three operations which we study in this thesis. The first is the *window search* operation. It finds all the point objects inside a given search window. The second operation is the *insert* operation, which inserts a point object into the index structure. The *delete* operation removes a point object from an index structure. These three operations are designated as follows:

$P.Search(W, R)$ Search the index structure rooted at R for the objects *inside* the search window W .

$P.Insert(O, R)$ Insert the object O into the index structure rooted at R .

$P.Delete(O, R)$ Delete the object O from the index structure rooted at R .

3.1.2 Rectangular Data

For rectangular data, we also have the above three operations. The *insert* and *delete* perform the same functions as before. However, the *window search* operation is different. Rather than finding objects enclosed by a given search window, it finds the objects which overlap with the window. The three operations are designated as follows:

$R.Search(W, R)$ Search the index structure rooted at R for the objects *overlapping* the search window W .

$R.Insert(O, R)$ Insert the object O into the index structure rooted at R .

$R.Delete(O, R)$ Delete the object O from the index structure rooted at R .

3.2 Locking Protocol

As in any database system which uses the pessimistic approach, a GIS may need to have a locking mechanism to permit concurrent access to spatial objects. To allow orderly access to a spatial index structure, we normally use three types of locks on the nodes of index structures.

A ρ lock is a read/share lock and it can be shared with other read operations¹ accessing the same node. Unlike a ρ lock, an ϵ lock is not compatible with any other ϵ or ρ lock. Recently, Otoo has suggested to use the “warning lock,” called the ω lock, for update operations to improve concurrency [Oto90]. We use this type of lock for the insert and delete operations in different spatial index structures. The compatibility among the three locks we use is shown in Figure 3.1. An ω lock allows its holder a read but not write access. However, unlike a ρ lock, an ω lock is not compatible with another ω lock. For all the index structures we study, we require that all locks be requested in a top-down, left-to-right (among the children of a node) fashion. This order of lock acquisition guarantees deadlock freedom. A ρ lock (ϵ lock) on a data item I is denoted by $\rho(I)$ ($\epsilon(I)$).

3.3 Definitions

To simplify the subsequent discussions, we now introduce a few short-hand notations.

<i>REGION(P)</i>	The region of a region node P .
<i>MBR(P)</i>	The MBR associated with node P .
<i>PARENT(P)</i>	Node identifier or pointer referencing the parent of P .
<i>FULL(P)</i>	Predicate indicating if node P is full.
<i>MIN(P)</i>	Predicate indicating if node P has the minimum number of entries.
<i>UNDERFL(P)</i>	Predicate indicating if node P has underflowed.
<i>DELETED(P)</i>	Predicate indicating if node P has been marked as deleted.
<i>PUSH(P,Q)</i>	Push node P onto stack Q (or append P to string Q).
<i>POP(Q)</i>	Remove and return the first element of Q . If Q is empty, return <i>NULL</i> .
<i>NEXT(P)</i>	Return the right sibling of P by using a link pointer.
<i>FOLLOW(P,Q)</i>	Return the next node following P in Q .

¹An operation, such as insert may carry out read operations while visiting nodes in an index structure.

FREE(Q) Empty queue Q .
POINT(O) Return the point location of object O .
NULL Null referencing pointer.

Chapter 4

Point Data

4.1 B⁺-Tree

The first point index structure we study in this chapter is the B⁺-tree [Com79]. A B⁺-tree of order M is a balanced tree such that each node, except the root, has at most M children and has at least m children for $m = \lceil M/2 \rceil$. The leaf nodes contain the actual key values and the internal nodes are used to direct the search for a key value. All leaf nodes are at the same level. An example of a B⁺-tree is shown in Figure 4.2. The concurrency control algorithms for the B⁺-tree and its variants have been studied extensively for many years. Unfortunately, the direct use of a B⁺-tree for point data in the 2 or higher dimensional space is difficult, because, in general, there is no good way of ordering the data points linearly, while preserving proximity. Some transformation techniques [Hin85, SeK88] have been suggested to associate a data point location with a scalar value and store the value into a B⁺-tree. We first review some of them and adopt the technique best suited to our purpose.

4.1.1 Point Transformation

Let \mathcal{D} be the given 2-dimensional data space. We assume without loss of generality that the coordinates of each data point are integers in the range $[0, n]$ for some n , and that a fixed number ($b = \lceil \log n \rceil$) of bits are used to represent them. Thus, a point (x, y) has the representation $(x_1 x_2 \dots x_b, y_1 y_2 \dots y_b)$, where the leading 0's, if any, are explicitly shown.

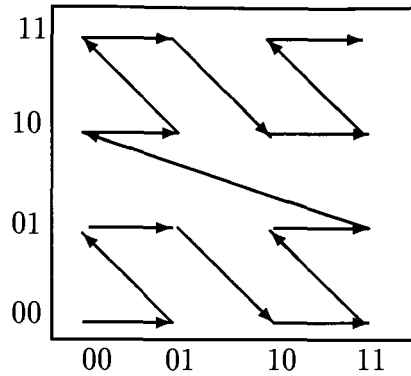


Figure 4.1: Z-ordering mapping.

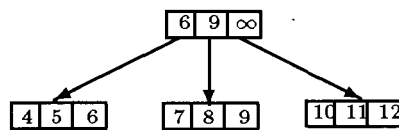


Figure 4.2: A B⁺-tree.

We need some one-to-one transformation

$$\psi : \mathcal{D} \rightarrow \mathcal{R},$$

where \mathcal{R} is a range of scalar values of the form $[0, r]$ for some integer r . One of the most popular such transformations is *bit-interleaving* [10] given by

$$\beta(x_1 x_2 \dots x_b, y_1 y_2 \dots y_b) = y_1 x_1 y_2 x_2 \dots y_b x_b.$$

In this case, $r \leq 2^{2b} - 1$. The transformation β is also known as the *z-ordering* (see 4.1).

We now introduce a measure of goodness for transformations. For a rectangle R in \mathcal{D} , let $|R|$ denote the number of “grid points” in R , and define $|\psi(R)|$ by

$$|\psi(R)| = \max\{\psi(x, y) \mid (x, y) \text{ in } R\} - \min\{\psi(x, y) \mid (x, y) \text{ in } R\}.$$

We use $|\psi(R)|/|R|$ as our measure of goodness. We clearly have $|\psi(R)|/|R| \geq 1$. Obviously, we want to use a transformation ψ such that $|\psi(R)|/|R|$ equals 1 for most R 's. Unfortunately, for the *z-ordering* β , $|\beta(R)|/|R|$ can be much larger than 1 for some R 's.

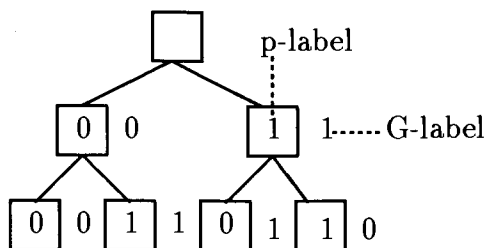


Figure 4.4: A p-G tree ($b = 2$).

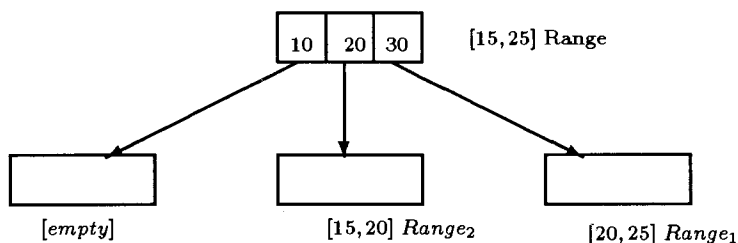


Figure 4.5: Decomposition of a query range.

be 11 (see Figure 4.4).

For insert and delete operations, we first transform the point by the transformation ψ_G introduced in above. The transformation is assisted by pre-constructing two p-G trees of height b and $2b$, respectively. We use a p-G tree of height b ($2b$) for pre- (post-) transformation. However, for a *window query*, we must transform the *search window* W into a linear *range* in order to find the point objects in the given window. After the ψ_G transformation, the query range may cover areas which are outside the given search window as we see below.

Given a search window W , let $Low = \min\{\psi_G(x, y) \mid (x, y) \text{ is a point in } W\}$ and $High = \max\{\psi_G(x, y) \mid (x, y) \text{ is a point in } W\}$. We can then represent W by the range $[Low, High]$. Suppose a point object in a leaf node is at position (a, b) . If $Low \leq \psi_G(a, b) \leq High$, we need to test whether it actually lies within W .

Our concurrency control protocol, BT, for B^+ -trees is implemented by a collection of procedures, for search, insert, and delete operations, as well as those for tree-reorganization. In the following sections, we discuss them in detail.

4.1.2 Search

At the beginning of a search operation, we need to compute the maximum scalar value (*High*) and the minimum scalar value (*Low*) of the given search window W as explained before. Let the four corners of W have the coordinates (X_l, Y_l) , (X_h, Y_l) , (X_l, Y_h) , and (X_h, Y_h) , where $X_l \leq X_h$ and $Y_l \leq Y_h$. In order to represent the range of x , i.e., $[X_l, X_h]$, we construct the $[X_l, X_h]$ -tree as follows. From the p-G tree of height b , remove all nodes and edges that lie to the left of the X_l -path and those lying to the right of the X_h -path. It is easy to see that any path in the $[X_l, X_h]$ -tree from the root to a leaf is an x -path for some x in the range $[X_l, X_h]$,

Next, in order to compute $\psi_G(x, y)$, where $X_l \leq x \leq X_h$ and $Y_l \leq y \leq Y_h$, we make use of three trees, the $[X_l, X_h]$ -tree, the $[Y_l, Y_h]$ -tree, and a p-G tree of height $2b$. For a given point (x, y) , let g_1, g_2, \dots, g_b be the G-labels of the x -path in the $[X_l, X_h]$ -tree, and f_1, f_2, \dots, f_b be the G-labels of the y -path in the $[Y_l, Y_h]$ -tree. We then traverse the p-G tree of height $2b$, starting at its root, visiting its child node with G-label = f_1 , then *its* child node with G-label = g_1 , then *its* child node with G-label = f_2 , etc. The sequence of the p-labels of the visited nodes is the desired value $\psi_G(x, y)$. Note that there is a one-to-one correspondence between a path in the p-G tree and a pair of x -path and y -path.

Finally, with this preparation, in order to compute *High*, we can proceed as follows. Note first that, the brute force method would generate $\psi_G(x, y)$ for all (x, y) such that there is an x -path (y -path) in the $[X_l, X_h]$ -tree ($[Y_l, Y_h]$ -tree), and set *High* to the largest generated value. We reverse this procedure by first traversing the p-G tree of height $2b$, trying to make $\psi_G(x, y)$ as large as possible. This can be accomplished by staying as far to the right of the p-G tree as possible, within the constraint that the corresponding x -path and y -path must exist in the $[X_l, X_h]$ -tree and the $[Y_l, Y_h]$ -tree, respectively.

A similar method is used to find *Low* except that at each node in the p-G tree of height $2b$, we try to move as far left as possible. We can show that both *High* and *Low* can be computed in $O(b)$ time, once the above three trees have been constructed.

A range query starts at the root of the B^+ -tree and descends down to the leaf level. At each visited node, the query range is compared to the index values and may be decomposed into several sub-ranges. In Figure 4.5, a query range is decomposed into two sub-ranges. To implement lock-coupling, at each visited node, a ρ lock on it must be acquired and held until the ρ locks on the child nodes to be visited are acquired (see Figure 4.6). This is to

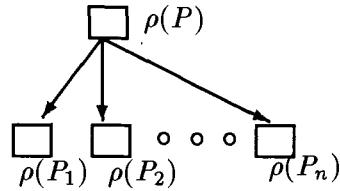


Figure 4.6: Search operation using the lock-coupling method.

avoid any change to the current node before the locking of those child nodes is completed. To implement this rule, we introduce a lockable variable, *Count*, which is initialized to the number of subsearch operations initiated at a node. It is decremented by one whenever a child node is ρ -locked. When it reaches 0, the ρ lock on the parent is released. At the leaf nodes, all point objects whose associated ψ_G values are within the query range are returned if they are also within W .

More formally, $Search(W, P)$ is carried out by calling $Bt.Search(W, P, Parent, Count, Low, High)$ given below, after setting $Count := 0$, $Parent = NULL$, $P := R$, where R is the root of the tree, and Low and $High$ to the minimum and maximum scalar values associated with W , respectively. *Count* in the following procedure is a call-by-reference parameter associated with *Parent*.

$Bt.Search(W, P, Parent, Count, Low, High)$

1. Acquire $\rho(P)$.
2. **if** $Parent \neq NULL$ (i.e., $P \neq root$) **then**
 - Acquire $\epsilon(Count)$.
 - Decrement *Count* by one.
 - **if** ($Count = 0$) **then** release $\rho(Parent)$.
 - Release $\epsilon(Count)$.
3. **if** P is a non-leaf node **then**
 - Let $\{E_i = (V_i, P_i) \mid i = 1, \dots, k\}$ be the entries in P such that $[V_i + 1, V_{i+1}] \cap [Low, High] = [Low_i, High_i] \neq \phi$
 - **if** ($k > 0$) **then**

```

    - MyCount := k.
    - For all entries  $E_i$  found above, if any, continue the search in parallel, invoking
      Bt.Search( $W, P_i, P, MyCount, Low_i, High_i$ ).
  else release  $\rho(P)$ .

else

  • For all points inside  $W$ , return their object ids.
  • Release  $\rho(P)$ . □

```

4.1.3 Insert

The implementation of an insert operation is very similar to that in [BaS77, Oto90]. There are two phases for an insert operation. During the first phase, we locate the leaf node to which to add the new object, and put it there. The path in the tree we take is called the **insertion path**. During the downward traversal, some full nodes¹ are ω -locked to avoid updates by other operations. The locked nodes will form a subpath at the leaf and of the insertion path.

The first phase is carried out by the procedure *Bt.Select* which locates the leaf node. If O is the point object to be inserted at (x, y) , a new entry $E_O = (\psi_G(x, y), O)$ will be added to the selected leaf node. After the point object is added, the ϵ -lock on the leaf node is downgraded to an ω -lock to avoid any possible deadlock with search operations.

In the second phase, the insert operation backtracks along the insertion path to reorganize the tree. The *insert* operation invokes the procedure *Bt.CleanUp* to reorganize the tree, if necessary.

More formally, *Insert*(O, P) under the lock-coupling method is carried out as follows, where O is the point object to be inserted, P is set to the root of the tree initially and $E_O = (\psi_G(x, y), O)$ is the new entry added to the selected leaf node.

Bt.Insert(O, P)

1. $\pi := \Lambda$ (empty string).
2. $L := Bt.Select(O, P, NULL, \pi)$, /* This procedure returns a ω -locked leaf node, L , in which to place object O . π is changed by call-by-reference and, on return, gives a stack of unsafe nodes, which forms a subpath of the insertion path. */

¹Nodes which have the maximum number of entries.

3. Add entry E_O to node L .
4. Downgrade $\epsilon(L)$ to $\omega(L)$.
5. $Bt.CleanUp(L, \pi)$. □

When $Bt.Select(O, P, Parent, Path)$ given below is recursively called, $Parent$ is ϵ -locked. On exit, P is ω -locked and $Parent$ is either placed on $Path$ with an ω -lock or $\epsilon(Parent)$ has been released. At each step, say at node P , $Bt.Select$ first acquires an ϵ -lock on the node and finds out the next node (P_c) to be visited. The ϵ -lock on P is not released until it finishes acquiring the ϵ -lock on P_c . This method prevents any other operation from “overtaking” an insert operation. In other words, no other operation which locks the root of the tree after the insert operation can lock a descendant node of P .

$Bt.Select(O, P, Parent, Path)$

1. Acquire $\epsilon(P)$.
2. **if** $Parent \neq NULL$ **then**
 - **if** $\neg FULL(P)$ **then**
 - Release $\epsilon(Parent)$.
 - Release all ω -locks on nodes in $Path$.
 - $FREE(Path)$.
 - else**
 - Downgrade $\epsilon(Parent)$ to $\omega(Parent)$.
 - $PUSH(Parent, Path)$.
3. **if** P is a non-leaf node **then**
 - Find an entry E_s in P , where $V_{s-1} < \psi_G(POINT(O)) \leq V_s$.
 - Continue the selection with $Bt.Select(O, P_s, P, Path)$.
- else** returns P . □

The procedure $Bt.CleanUp$ given below is used to reorganize the tree after an object is inserted into the tree. When it is called, unless $L = NULL$, L is ω -locked, and except when $PARENT(L) = NULL$ (i.e., $L = root$) or $Path = \Lambda$, $PARENT(L)$ is also ω -locked.

$Bt.CleanUp(L, Path)$

1. **if** $((L = NULL) \text{ or } (\neg FULL(L)) \text{ or } (Path = \Lambda))$ **then**
 - Release $\omega(L)$.
 - Return.
2. $Parent := POP(Path)$.
3. **if** $(Parent = NULL)$ **then**
 - Create a new root,
 - ϵ -lock the new root,
 - Make it $Parent$.
 - Add L into $Parent$.
4. Acquire $\epsilon(Parent)$.
5. Acquire $\epsilon(L)$.
6. Split L into L and L' .
7. Add L' to $Parent$. /* We provide an extra entry in each node to accommodate temporary overflow. Hence L' can be added to $Parent$ even when $Parent$ is $FULL$. */
8. Release $\epsilon(L)$.
9. Downgrade $\epsilon(Parent)$ to $\omega(Parent)$.
10. $Bt.CleanUp(Parent, Path)$. □

4.1.4 Delete

$Delete(O, P)$ can be carried out in a manner similar to the insert operation. The first phase is to locate the object to be removed with procedure $Bt.Find$ and remove the object. This procedure is similar to $Bt.Select$ for an insert operation, except that it ω -locks the nodes which have underflowed. The second phase reorganizes the tree if an underflow occurs as a result of removing the object.

There are two arguments for $Bt.Delete$, where O is the object to be deleted and P is the root of the tree initially.

$Bt.Delete(O, P)$

1. $\pi := \Lambda$ (empty string).
2. $L := \text{Bt.Find}(O, P, \text{NULL}, \pi)$, /* This procedure returns an ω -locked leaf node, L , in which object O is referenced. π is changed by call-by-reference and, on return, gives a stack of under-utilized nodes, which forms a subpath of the deletion path. */
3. Remove entry E_O from node L .
4. Downgrade $\epsilon(L)$ to $\omega(L)$.
5. $\text{Bt.Condense}(L, \pi)$. □

When the following procedure $\text{Bt.Find}(O, P, \text{Parent}, \text{Path})$ is recursively called, Parent is ϵ -locked. On exit, P is ω -locked, and Parent is either placed on Path with an ω -lock or $\epsilon(\text{Parent})$ has been released.

$\text{Bt.Find}(O, P, \text{Parent}, \text{Path})$

1. Acquire $\epsilon(P)$.
2. **if** $\text{Parent} \neq \text{NULL}$ **then**
 - **if** not $\text{MIN}(P)$ **then**
 - Release $\epsilon(\text{Parent})$.
 - Release all locks on nodes in Path .
 - $\text{FREE}(\text{Path})$.
 - else**
 - Downgrade $\epsilon(\text{Parent})$ to $\omega(\text{Parent})$.
 - $\text{PUSH}(\text{Parent}, \text{Path})$.
3. **if** P is a non-leaf node **then**
 - Find an entry E_s in P , where $V_{s-1} < \psi_G(\text{POINT}(O)) \leq V_s$.
 - $\text{Bt.Find}(O, P_s, P, \text{Path})$.
- else** returns P . □

When the following procedure is called, unless $L = \text{NULL}$, L is ω -locked, and except when $\text{PARENT}(L) = \text{NULL}$ (i.e., $L = \text{root}$) or $\text{Path} = \Lambda$, $\text{PARENT}(L)$ is ω -locked.

$\text{Bt.Condense}(L, \text{Path})$

1. **if** ($L = NULL$) or (not $UNDERFL(L)$ or ($Path = \Lambda$)) **then**
 - Release $\omega(L)$.
 - Return.
2. $Parent := POP(Path)$.
3. **if** ($Parent = NULL$) **then**
 - Acquire $\epsilon(L)$.
 - **if** (L has no child) **then**
 - Remove L .
 - Set root of the tree to $NULL$.
 - Release $\epsilon(L)$.
 - Return.
4. Acquire $\epsilon(Parent)$.
5. Let $Parent$ have k children P_1, \dots, P_k , including L .
6. Acquire $\epsilon(P_i)$ for $i = 1, \dots, k$.
7. Move the entries of L to its siblings.
8. Remove the entry in $Parent$ which references L .
9. Mark L as deleted.
10. Release $\epsilon(P_i)$ for $i = 1, \dots, k$.
11. Downgrade $\epsilon(Parent)$ to $\omega(Parent)$.
12. $Bt.Condense(Parent, Path)$. □

4.1.5 Correctness of the Operations

The two theorems in this subsection prove the correctness of the concurrency control protocol BT for the B^+ -tree, which consists of a set of procedures described above.

Consider a scenario where S_1 and S_2 are two different search operations. Suppose S_1 and S_2 both visit node A at the same time. At node A of the tree, S_1 is divided into subrange queries S_{11} and S_{12} , which will take two different paths down and visit the leaf nodes P_1 and P_2 , respectively, (see Figure 4.7). Similarly, S_2 is divided into S_{21} and S_{22} .

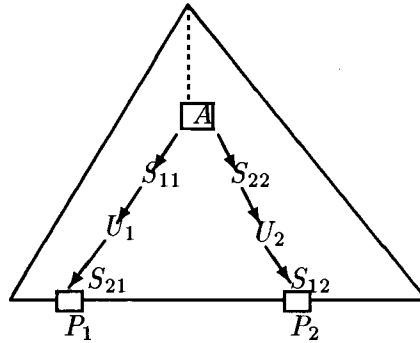


Figure 4.7: Multiple sub-searches.

If the two search operations execute at different speeds, then S_{21} overtakes S_{11} , while S_{12} overtakes S_{22} . Furthermore, let there be two update operations (transactions), U_1 and U_2 . U_1 traverses along the same path as S_{21} and U_2 traverses along the same path as S_{12} . At the leaf level, U_1 modifies P_1 and U_2 modifies P_2 . If overtaking between search and update operations is permitted, U_1 (U_2) may work on P_1 (P_2) before S_{11} (S_{12}) but after S_{21} (S_{22}). When this happens, at P_1 the execution is seen as $[S_2][U_1][S_1]$, while at P_2 the execution is seen as $[S_1][U_2][S_2]$. Hence, S_1 and S_2 see different results even if they have the same search window and the executions are not serializable [BHG87].

For a set of search, insert and delete operations executed on a B^+ -tree, B , we can denote them by $S_i(W)$, $I_j(O)$, and $D_k(O)$, respectively, where W is a search window and O is an object. The subscripts are used to distinguish among different operations. So, when we need to discuss only the type of operation, we often use $S(W)$, $I(O)$, and $D(O)$. An execution is defined by a “precedes” relation among them, denoted $<$, based on the order in which they lock the root of B . Given an execution, \mathcal{E} , we say that an object O is **available** to a search operation $S(W)$ in \mathcal{E} , such that $O \in W$, if the last update operation, if any, accessing O that precedes $S(W)$ is not $D(O)$ and either O was in B before the execution of the set of operations under discussion or there is an $I(O) < S(W)$. With the above definitions, we say that a protocol \mathcal{P} on an index structure is **correct** if it satisfies the following two conditions.

1. \mathcal{P} is deadlock free, and
2. In any execution \mathcal{E} that \mathcal{P} generates, each search operation $S(W)$ returns object O iff

O is available to it in \mathcal{E} .

In order to prove the second condition, it is sufficient to show the following:

- (a): Conflicting operations accessing the same object do not overtake each other.
- (b): If an object O is available to $S(W)$, then at least one subsearch maintains a window W' such that $O \in W'$, until O is returned.

In the rest of this section, an operation means an insert, delete or a subrange query on a leaf node. We sometimes refer to insert ($I(O)$) and delete operations ($D(O)$) as **update** operations ($U(O)$). Furthermore, each operation obtains a timestamp when it first accesses the root. We will also call a node **unsafe** if (with the currently available information) there is a possibility that the node will be modified in the second phase of an update operation. We next introduce two definitions for later discussions.

Scope: The scope of an insert operation is the subtree rooted at the first currently unsafe node along its insertion path.

Overtake: Suppose S_1 and S_2 are two different operations. S_2 overtakes S_1 if S_1 is younger than S_2 and S_2 visits nodes at deeper levels earlier than S_1 along the same access path.

Lemma 4.1 *With the lock-coupling protocol, a search operation cannot overtake an update operation. Similarly, an update operation cannot overtake any other operation.*

Proof: When two update operations visit the same node, because ω -locks are incompatible with each other, the second update operation has to wait and cannot overtake the first one.

Consider the case where a search operation S and an update operation U trying to lock a node P at the same time. If S acquires a ρ -lock on P first, U , which requests an ϵ -lock will wait until S releases its ρ -lock on P , which occurs after S has acquired all the ρ -locks it needs on P 's child nodes. Hence, U cannot overtake S . If U acquires its ϵ -lock first, it will hold the lock until it has acquired the ϵ -lock on the node next visited. U may then release the lock on P or downgrade the lock to an ω -lock. In both cases, S visits the node after U has finished its work at P . Hence a search operation cannot overtake an update operation.

□

Theorem 4.1 *Protocol BT is deadlock and livelock free.*

Proof: Assume that a set of operations is deadlocked under the protocol BT, and let Δ be the set of all deadlocked operations. We derive a contradiction out of this assumption. Let T be the oldest operation in Δ .

- **a:** T is a search operation. Since T uses ρ -lock-coupling, no update operation can overtake T . Therefore, no node that T is trying to lock is ϵ -locked by an update operation in Δ . T can thus always acquire a ρ lock it needs, and cannot be blocked forever, a contradiction.
- **b:** T is an insert operation in its first phase. Since T uses ϵ -lock-coupling, no operation can overtake T . Thus, T cannot be blocked forever, a contradiction.
- **c:** T is an insert operation in its second phase, executing *Bt.CleanUp*. In this case, T is trying to upgrade an ω -lock to an ϵ -lock. From Lemma 4.1, T cannot be in conflict with any other update operation to upgrade the lock because no younger update operation can be within its scope. A younger search operation may have placed a lock on the node which T is trying to acquire, but it will eventually terminate because of the order of lock acquisition/upgrade (top-down and left-to-right fashion). Note that individual subsearch operations belonging to a search operation can proceed independently, since they are executed in parallel. Thus, T cannot be blocked forever by search operations. Hence, a contradiction to the assumption.
- **d:** T is a delete operation in its first phase. The argument is the same as in (b).
- **e:** T is a delete operation in its second phase. The argument is the same as in (c).

In general, lock starvation is possible when an operation continues to fail in acquiring a lock on a node due to unfortunate timing. This can be avoided by having a “fair” locking policy such as using the request times of a lock to be the order of granting the lock. \square

Lemma 4.2 *In the protocol BT, no search operation will miss an available object.*

Proof: To prove the lemma, we will only need to consider the situation where there is mix of search and update operations.

Let U_1 be an update operation in its first phase and S_1 be a search operation traversing the same path together. During the downward traversal of U_1 , it does not modify any

internal nodes except a leaf node, say L . Hence, S_1 will always arrive at L . If S_1 acquires a ρ -lock on L first, then U_1 will need to wait. On the other hand, S_1 will wait when U_1 ϵ -locks L first. In either cases, S_1 will see the available objects in L correctly.

We now consider the case where there is a cleanup or a condense operation backtracking along the same path that S_1 uses. Suppose S_1 is visiting node P . It will first acquire a ρ -lock on P and then determine the nodes to visit next. Let P_c be one of the nodes in P_1, \dots, P_k to be visited by S_1 next. If U_1 is in its second phase, trying to modify P_c , then P_c can only be changed if U_1 has acquired an $\epsilon(P_c)$. If this happens, S_1 will wait until U_1 modifies and releases the locks. S_1 still correctly arrives at P_c because P_c cannot be split or removed, since S_1 holds a ρ -lock on P . U_1 might have changed the content (e.g., the number of entries) in P_c . However, this does not affect the results of S_1 because U_1 has already inserted or deleted an object at a leaf node. If S_1 gets a ρ -lock on P_c before U_1 ϵ -locks it, then S_1 will work on P_c as intended and U_1 will wait until S_1 finishes. Hence, the search operation will not miss any object which is available to it. \square

Theorem 4.2 *The protocol BT is correct.*

Proof: From Lemma 4.1, Lemma 4.2 and Theorem 4.1, we can conclude that the protocol BT is correct. \square

4.2 R-Tree

In our concurrency control algorithms for R-tree, we also use the lock-coupling method. Associated with each leaf node of an R-tree is a **minimum bounding rectangle (MBR)** that encloses all the points pointed to by the node. Similarly, an MBR is associated with each internal node, which encloses the MBRs associated with its child nodes. Thus, the locations of the points referenced by a leaf node L are inside the MBRs associated with all the nodes on the path from L to the root of the tree. Each node of an R-tree contains a number of entries. Each entry in a node contains a MBR and a reference pointer to another node. In an R-tree, the MBRs associated with different nodes may overlap, so that an object may be spatially contained in the MBRs associated with several nodes, yet it can only be represented by one leaf node. This implies that processing a query often requires traversing several search paths before ascertaining the presence or absence of a particular object. An example of an R-tree with the corresponding MBRs is shown in Figure 4.8.

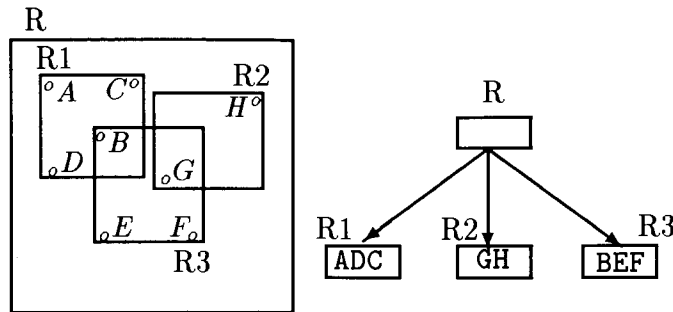


Figure 4.8: An R-tree.

Our concurrency control protocol, RT, for R-trees is implemented by a collection of procedures, for search, insert, and delete operations, as well as those for tree-reorganization. In the following sections, we discuss them in detail.

4.2.1 Search

A search starts at the root of an R-tree and descends to the leaf level to find objects in a given search window. Along its way, it acquires ρ locks on the visited nodes. The ρ -lock on the parent is released when the locks on its children to be visited have been acquired. We use the lockable variable, *Count*, for the same purpose as for the B^+ -tree. To find which subtrees to visit, at each node, a search operation compares the search window with the MBRs associated with the child nodes.

More formally, $Search(W, R)$ on an R-tree is carried out by calling $Rt.Search(W, P, Count)$ given below, where W is the search window and $Count$ is a call-by-reference parameter. Initially, P points to the root of the tree and $Count := 0$.

$Rt.Search(W, P, Count)$

1. Acquire $\rho(P)$.
2. **if** $PARENT(P) \neq NULL$ (i.e., $P \neq root$) **then**
 - Acquire $\epsilon(Count)$.
 - Decrement $Count$ by one.

- **if** ($Count = 0$) **then** release $\rho(PARENT(P))$.
- Release $\epsilon(Count)$.

3. **if** P is a non-leaf node **then**

- Let $\{E_i = (MBR(P_i), P_i) \mid i = 1, \dots, k\}$ be the entries in P such that $MBR(P_i) \cap W \neq \phi$.
- **if** ($k > 0$) **then**
 - $MyCount := k$.
 - For all entries E_i found above, continue the search in parallel, invoking $Rt.Search(W \cap MBR(P_i), P_i, MyCount)$.
- else** release $\rho(P)$.

4. **if** P is a leaf node **then**

- For all objects inside W , return their object ids.
- Release $\rho(P)$.

□

4.2.2 Insert

As with the insert operation on the B^+ -tree, we carry out an insert operation in two phases. During the first phase, we locate the leaf node to which to add the new object and insert the object there. The path in the tree we take is called the **insertion path**. In the second phase, we may have to perform some maintenance operations by tracing the insertion path back to

1. reorganization of the tree, and/or
2. enlarge the MBRs of some nodes on the path.

If a node visited in the first phase is full, then there is a possibility that it may be split during the second phase. If a split does occur, the parent of the split node will have a new entry. This information must be available to the subsequent operations. We can use a solution similar to the one used in the B^+ -tree. Namely, in the first phase of an insert operation, we ω -lock the first full node encountered in the insertion path, and its parent, if

any. Thereafter, we ω -lock consecutive full nodes. Once a non-full node is encountered along the insertion path, we release all the existing ω -locks and restart the above procedure from scratch. So, when we reach a leaf, we will have ω -locked a string of zero or more nodes at the end of the insertion path. If this string is non-empty, then it starts with a non-full node (with one exception mentioned below) and is followed by one or more full nodes. During the second phase, splitting will propagate upwards to that only one ω -locked non-full node, but will terminate there. If the root node is full, it is possible that this string consists only of full nodes, and a new root will be created in the second phase.

Let us call a node **unsafe** if (with the currently available information) there is a possibility that maintenance operations in the second phase may reach that node. What we described above can be restated by the following rule:

Safety rule: In the first phase, ω -lock the unsafe nodes along the insertion path, and release ω -locks from nodes as they become safe (i.e., not unsafe.)

There is another reason why a node may be unsafe; it is due to the possibility of maintenance action 2. mentioned above. In the first phase, we change the MBRs of some nodes, if the point to be inserted is not within its current MBR. If the MBR of a node P is updated and the ω -lock on P is released, then P 's MBR may be changed by a subsequent delete operation, which may cause a later search operation to have an incorrect answer. This problem is solved by maintaining an ω -lock on the node until the second phase of an insertion is completed. Incidentally, observe that, if a node's MBR is expanded, then all its descendants' MBRs along the insertion path will also be expanded. The safety rule we stated above covers all unsafe nodes.

As an illustration, consider Figure 4.9, where an insert operation, I_1 , is visiting node B in its first phase. It first acquires an ϵ -lock on B to prevent other operations from changing B . It will then visit node C . After acquiring an ϵ -lock on node C , we expand B 's MBR, if it is necessary to reflect I_1 . If C is full, we downgrade the ϵ -lock on B to an ω -lock; otherwise we simply release it.

During the second phase, after inserting a new point object into the selected leaf node, the insert operation backtracks along the insertion path to split nodes if necessary. In Figure 4.9, suppose that node D overflows, after the new point object has been added to a leaf node. Before splitting D , we upgrade the ω -locks on nodes C and D to ϵ -locks. If D is a non-leaf node, we also ϵ -lock D 's children, because their parent pointers need to be

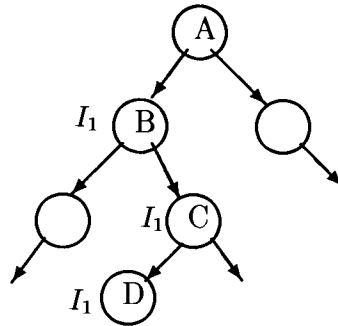


Figure 4.9: Insert operation using the lock-coupling method.

updated due to the split. As stated earlier, these locks should be acquired in the top-down, left-to-right order. After the split, we release the ϵ -locks on D and its children, if any, and downgrade the ϵ -lock on C to an ω -lock. We continue the second phase of an insert operation, level by level, until we reach the first unsafe node on the insertion path.

More formally, an insert is carried out by $Insert(O, P)$ given below, where O is the object to be inserted, P is the root of the tree initially, and $E_O = (POINT(O), O)$ is the entry referencing the object to be added.

Rt.Insert(O, P)

1. $\pi := \Lambda$ (empty string).
2. $L := Rt.Select(O, P, \pi)$, /* This procedure returns an ϵ -locked leaf node, L , in which to place entry E_O . π is changed by call-by-reference and, on return, gives a stack of unsafe nodes, which forms a subpath of the insertion path. */
3. Add entry E_O . to node L^2
4. Downgrade $\epsilon(L)$ to $\omega(L)$.
5. *Rt.CleanUp*(L, π).

□

In the following procedure, $Rt.Select(O, P, Path)$, we define $MBR(NULL) = \phi$. Note that, when this procedure is recursively called, $PARENT(P)$ is ϵ -locked, and on exit, P is ϵ -locked. $PARENT(P)$ is either placed on $Path$ with an ω -lock, or its ω -lock has been

²If L is full, temporary overflow is allowed and the splitting will be done in *Rt.CleanUp*.

released. The variable *unsafe* is used to indicate the safety status of a node. The node is safe when it has the value 0. It has the value 1 if the node is full, and the value 2 if the new object is outside the MBR of the node. It equals 3 if the node is full and the new object is outside its MBR.

Rt.Select(*O*, *P*, *Path*)

1. Acquire $\epsilon(P)$.
 2. **if** *PARENT*(*P*) \neq *NULL* **then**
 - *unsafe* := 0.
 - **if** *FULL*(*P*) **then** *unsafe* := 1.
 - **if** *O* is outside *MBR*(*PARENT*(*P*)) **then**
 - *unsafe* := *unsafe* + 2.
 - Update *MBR*(*P*) to include *O*.
 - **if** *unsafe* \geq 1 **then**
 - *PUSH*(*PARENT*(*P*), *Path*).
 - Downgrade ϵ (*PARENT*(*P*)) to ω (*PARENT*(*P*)).
 - **if** (*unsafe* = 0) **then** release ϵ -lock on *PARENT*(*P*).
 - **if** ((*unsafe* = 0) and (*Path* \neq Λ)) **then** release the locks on all nodes in *Path*, and *FREE*(*Path*).
 3. **if** *P* is a non-leaf node **then**
 - Find an entry $E_s = (\text{MBR}(P_s), P_s)$ in *P*, where *MBR*(*P_s*) requires the smallest enlargement to include *O*. Resolve ties by choosing E_s with the smallest *MBR*(*P_s*).
 - Continue the selection with *Rt.Select*(*O*, *P_s*, *Path*).
- else** return *P*.

□

When the following procedure is called, unless $L = \text{NULL}$, *L* is ω -locked, and except when *PARENT*(*L*) = *NULL* (i.e., $L = \text{root}$) or *Path* = Λ , *PARENT*(*L*) is ω -locked.

Rt.CleanUp(*L*, *Path*)

1. **if** $L = NULL$ **then** return.
2. **if** $OVERFL(L)$ **then**
 - **if** $PARENT(L)$ exists **then** upgrade $\omega(PARENT(L))$ to $\epsilon(PARENT(L))$ **else**
 - Create a new root,
 - ϵ -lock the new root,
 - Make it $PARENT(L)$.
 - Acquire $\epsilon(L)$.
 - Split L into L and L' .
 - Add entry $E_{P'}$ representing L' to $PARENT(L)$ and update its MBR.
 - **if** L and L' together have k children, $L_1 \dots, L_k$ **then**
 - acquire $\epsilon(L_i)$ for $i = 1, \dots, k$.
 - Set the parent pointer in L_i for $i = 1, \dots, k$.
 - Release $\epsilon(L_i)$ for $i = 1, \dots, k$.
 - Release $\epsilon(L)$.
 - Downgrade $\epsilon(PARENT(L))$ to $\omega(PARENT(L))$.
 - $L := POP(Path)$.
 - $Rt.CleanUp(L, Path)$.
- else**
 - Release all ω -lock on $Path$.
 - $FREE(Path)$.

□

4.2.3 Delete

We use two phases to carry out a delete operation. The first phase is to locate the object. This phase is almost identical to the search operation. (Note that the condition “ O is inside $MBR(P_i)$ ” should replace “ $MBR(P_i) \cap W \neq \phi$ ” in Step 3 of $Rt.Search$.) Internal nodes are ϵ -locked to avoid overtaking by other operations and the MBRs are not shrunk when a delete operation traverses down the R-tree. Once the object has been found, it is removed and the tree is reorganized in the second phase.

In the second phase, we backtrack the R-tree, if needed. We now face two problems. The first is how to deal with concurrent insertions and changing parent pointers. The delete operation needs to know how to backtrack to propagate the change up the tree. The second problem is the potentially conflicting lock requests with other updating operations. We need to ensure lock requests do not cause deadlocks.

To solve the first problem, we maintain another piece of information at each node of the R-tree, i.e., the parent pointer. The pointer of a node is updated whenever its parent is involved in splits or merges. The second problem is solved by ordering the lock requests and having the procedure *Rt.Delete* release all of its locks it holds from time to time. The ordering ensures delete operations are not in conflict with each other, and the releasing of locks permits insert operations to acquire locks which may be held by a delete operation.

During deletion, a node is not removed immediately because there can be other delete operations working on it. We use the “mark-and-remove” approach. If a node becomes empty as a result of removing an entry from it, then it is flagged as “deleted.” “Deleted” nodes are later garbage-collected periodically. To perform the garbage-collection, we can record the deletion time of a marked node and the starting time of each operation. We can remove a marked node when all the current active operations are started after the node’s deletion time. More formally, *Rt.Delete*(O, R) under the lock-coupling method is carried out as follows.

Rt.Delete(O, R)

1. $L := \text{Rt.Find}(O, R, 0)$ to locate the leaf node which contains object O .
2. Delete entry E_O which points to object O .
3. Downgrade $\epsilon(L)$ to $\rho(L)$.
4. *Rt.Condense*(L).

Rt.Find(O, P, Count)

1. Acquire $\epsilon(P)$.
2. **if** $\text{PARENT}(P) \neq \text{NULL}$ **then**
 - Acquire $\epsilon(\text{Count})$.
 - Decrement Count by one.

- **if** ($Count = 0$) **then** release $\epsilon(PARENT(P))$.

- Release $\epsilon(Count)$.

3. **if** P is a non-leaf node **then**

- Let $\{E_i = (MBR(P_i), P_i) \mid i = 1, \dots, k\}$ be the entries in P such that O is contained in $MBR(P_i)$.

- **if** ($k > 0$) **then**

- $MyCount := k$.

- For all entries E_i found above, continue the search in parallel invoking $Rt.Find(O, P_i, MyCount)$.

else release $\epsilon(P)$.

4. **if** P is a leaf node **then**

- **if** there is an entry $(POINT(P_i), P_i)$ such that $POINT(O) = POINT(P_i)$ and O is referenced by P_i **then** return P .

else release $\epsilon(P)$.

Rt.Condense(L)

1. **if** $DELETED(L)$ **then**

- Release $\rho(L)$.

- Return.

2. $P := PARENT(L)$.

3. Release $\rho(L)$.

4. **if** ($P = NULL$) **then** /* i.e., when L is the root */

- Acquire $\epsilon(L)$.

- Update the MBR of L .

- **if** (L has only one child) **then**

- Let L_c be the only child.

- Make L_c the new root of the tree.

- Mark L as deleted.

- Release $\epsilon(L)$ and return.
5. Acquire $\epsilon(P)$.
 6. Acquire $\epsilon(L)$.
 7. **if** ($P \neq \text{PARENT}(L)$) **then**
 - Release $\epsilon(L)$ and $\epsilon(P)$.
 - Acquire $\rho(L)$.
 - Goto Step 1.
 8. **if** ($\neg \text{UNDERFL}(L)$) and ($\text{MBR}(P - L)^3 = \text{MBR}(P)$) **then**
 - Release $\epsilon(L)$ and $\epsilon(P)$, and return.
 9. **if** ($\neg \text{UNDERFL}(L)$) **then**
 - Update the MBR of P .
- else**
- Acquire $\epsilon(L^j)$, where L^j 's are the m siblings of L .
 - **if** L is a non-leaf node **then**
 - Acquire $\epsilon(L_i)$, where L_i 's are the k children of L .
 - Merge the nodes by re-distributing L_i 's among L^j 's.
 - Update the MBR's of P and L^j 's.
 - Update the parent pointers in L_i 's.
 - Mark L as deleted.
 - **if** L is a non-leaf node **then** release $\epsilon(L_i)$ for $i = 1, \dots, k$.
 - Release $\epsilon(L^j)$ for $j = 1, \dots, m$.
10. Release $\epsilon(L)$.
 11. Downgrade $\epsilon(P)$ to $\rho(P)$.
 12. *Rt.Condense*(P).

□

³ $\text{MBR}(P - L)$ represents the new MBR when a child node L is removed from node P .

4.2.4 Correctness of the Operations

We prove the correctness of our concurrency control protocol RT for the R-tree by first showing it is deadlock-free. We then prove that a search operation can always find the available objects. In the following discussions, an *update operation* means either an insert or a delete operation. We start with the following lemma:

Lemma 4.3 *A search operation cannot overtake an update operation if they are traversing the same path. Similarly, an update operation cannot overtake any other operation.*

The proof is the same as in Lemma 4.1. □

Theorem 4.3 *Protocol RT is deadlock free.*

Proof: Similarly to the proof for the BT algorithms, we derive a contradiction by first assuming there is a deadlock. Let Δ be the set of all deadlocked operations and T be the oldest operation in Δ .

- **a:** T is a search operation. Since T uses ρ -lock-coupling, no update operation can overtake T . Therefore, no node that T is trying to lock is ϵ -locked by another update operation in Δ . T can thus always acquire a ρ lock it needs, and cannot be blocked forever, a contradiction.
- **b:** T is an insert operation in its first phase. Since T used ϵ -lock-coupling, no operation can overtake T . Thus, T cannot be blocked forever, a contradiction.
- **c:** T is an insert in its second phase, executing *Rt.CleanUp*. In this case, T is trying to upgrade an ω -lock to an ϵ -lock. From Lemma 4.3, T cannot be in conflict with any younger update operation because no update operation can be within its scope. A younger search operation may have placed a lock on the node which T is acquiring, but it will eventually terminate because of the order of lock acquisition/upgrade (top-down and left-to-right fashion). Note that individual subsearch operations belonging to a search operation can proceed independently, since they are executed in parallel. Thus, T cannot be blocked forever by search operations. Hence, a contradiction to the assumption.
- **d:** T is a delete operation in its first phase. The argument is the same as in (b).

- **e:** T is a delete operation in its second phase. In this case, T is trying to acquire ϵ -locks, say on node P and P 's child nodes. A younger operation (T_1) may have placed a lock on P , but it will eventually terminate. It is because of the order of lock acquisition and T does not hold any lock before successfully acquires the lock on P . Hence, T cannot be blocked forever. A contradiction to the assumption. \square

We like to point out that the *delete* algorithm does not prevent the possibility of *livelock* (where one operation runs indefinitely). This can happen when a delete operation backtracks and the parent of the currently visited node is kept changing (i.e., the delete operation is running very slow and the parent keeps on splitting). However, we believe that the situation is unlikely to happen in a practical system with the following ideas.

- In most systems, an operation is executed as a single process and all processes are run with comparable speed.
- Insert and delete operations occur infrequently when compared to search operations.
- In a short time interval, there is only a small number of new nodes created.

In the worse situation where operations do run at different speeds, we can introduce some addition mechanism to prevent livelock. One possibility is to assign lock acquisition priorities to each operation according to the “age”⁴ of the operation. This would guarantee that an operation will finish if it becomes the oldest.

Lemma 4.4 *In the protocol RT , no search operation will miss an available object.*

Proof: The proof is analogous to that for Lemma 4.2. When all operations are search operations, as there is no modification made to the tree, every search operation will see the available objects correctly.

We then consider when a search operation is traversing the path together with an update operation in its first phase. Let I_1 be an insert operation, S_1 be a search operation. If I_1 is younger (has later starting time) than S_1 , as overtaking is not possible by Lemma 4.3, I_1 will not affect the result of S_1 at all. If I_1 is older than S_1 , I_1 will expand the MBRs along the insertion path, if necessary. Hence, S_1 cannot miss the new object. If the update operation is a delete operation, U_1 , during the downward traversal of U_1 , it does not modify any nodes

⁴The elapsed time since an operation started.

except a leaf node, say L . Hence, S_1 will always arrive at L . If S_1 acquires a ρ -lock on L first, then U_1 will need to wait. On the other hand, S_1 will wait when U_1 acquires an ϵ -lock on L first. In the above cases, S_1 will see the available objects in L correctly.

We now consider the case where there is a cleanup or a condense operation backtracking along the same path that S_1 uses. Suppose S_1 is visiting node P . It will first acquire a ρ -lock on P and then determine the nodes to visit next. Let P_c be one of the nodes in P_1, \dots, P_k to be visited by S_1 next. If U_1 is in its second phase, trying to modify P_c , then P_c can only be changed if U_1 has acquired an $\epsilon(P_c)$. If this happens, S_1 will wait until U_1 modifies and releases the locks. S_1 still correctly arrives at P_c because P_c cannot be split or removed, since S_1 holds a ρ -lock on P . If U_1 is a cleanup operation, then the MBR of P_c cannot be modified and S_1 continues its work correctly. If U_1 is a condense operation, then the MBR of P_c might have been shrunk. However, this does not affect the results of S_1 because the adjustment of the MBR must have been done after U_1 's removal of an object which occupied the missing space. If S_1 gets a ρ -lock on P_c before U_1 locks it, then S_1 will work on P_c as intended and U_1 will wait until S_1 finishes. Hence, the search operation will not miss any object which is available. \square

Theorem 4.4 *The protocol RT is correct.*

Proof: From Lemma 4.3, Lemma 4.4, and Theorem 4.3, we can conclude that protocol RT are correct. \square

4.3 K-D-B Tree

The K-D-B tree [Rob81] partitions a cube in a k -dimensional space into non-overlapping regions to index point data. The tree is always balanced in the sense that the number of nodes on the path from the root node to any leaf node is the same. However, unlike B^+ -trees, 50% storage utilization of nodes cannot be guaranteed because of the splitting conditions. There are two types of nodes in a K-D-B tree: the *region node* and *point node*. A region node is an internal node, which contains entries of the form $(region, child_id)$, where *region* represents a region box in the space and *child_id* is the reference pointer to a child node. On the other hand, a point node is a leaf node, which contains a collection of $(object_id, location)$ pairs, where *object_id* is a reference to a point object and *location* gives the spatial location of the object.

In the previous section, the MBRs of entries within the same node of an R-tree were allowed to overlap. This resulted in the possibility of multiple search paths during data access. Unlike an R-tree, a K-D-B tree partitions the data space into non-overlapping regions. No pair of nodes at the same level of a K-D-B tree have overlapping MBRs. The no-overlap property allows the search operations to be more efficient, because it reduces unnecessary search paths. However, the same property also introduces new problems for the insert and delete operations.

The first problem is the *up-and-down splitting* of nodes in the tree. During the second phase of an insert operation, node splitting may have to be propagated not only up the tree (towards the root) but also down the tree (towards leaves). When a node is full and needs to be split, the split line chosen may cause its children to be split as well. Suppose in Figure 4.10, C is split into C (the left half) and C' (the right half). Because the split line crosses its children A and B , they need to be split as well. If the children of A and B intersect with the line, more splits will occur. We call these splits the *recursive splitting* of C . After finishing the splits for C , A and B , we may find C' full and more splitting may be required. Thus, several traversals of the K-D-B tree may be required before updating is completed. The recurring splits create extra complexities and result in poor performance. To solve the problem, we use the *forced splitting* strategy for the insert operations, to be explained later in more detail.

The second problem is related to the merging of entries within a node in the K-D-B tree. This problem occurs during the clean-up phase of a delete operation. For any entry in a node, its associated region is a rectangular region. If an entry is removed or the child node it references underflows, it needs to merge with other entries in the same node. However, the merging of regions is limited by the *joinability* property. For a given rectangle R with sub-rectangles, R_1, R_2, \dots, R_k , a joinable set of rectangles with respect to R_1 consists of a subset of these rectangles, including R_1 , whose union is also a rectangle. In Figure 4.11, there are four sets of joinable rectangles with respect to D . In the discussion on the delete operation, we will suggest to use the *minimal joinable set* of entries in a node together with splitting to rearrange the entries.

Our concurrency control protocol, KT, for K-D-B trees is implemented by a collection of procedures, for search, insert, and delete operations, as well as those for tree-reorganization. In the following sections, we discuss them in detail.

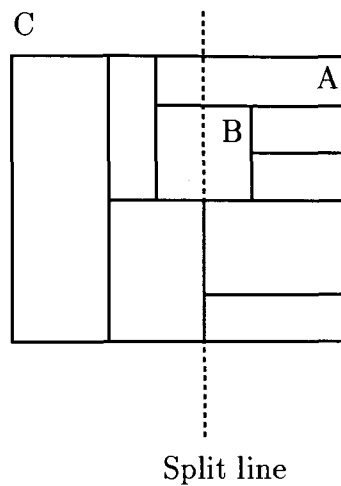


Figure 4.10: Recursive splitting.

4.3.1 Search

The search operation is implemented in the same way as that in the R-tree. It starts at the root of a tree and descends to the leaf level. More formally, $\text{Search}(W, R)$ is carried out by calling $\text{Kt.Search}(W, P, \text{Count})$ given below, where W is a search window, $\text{Count} := 0$, by setting P to the root of the tree.

$\text{Kt.Search}(W, P, \text{Count})$

1. Acquire $\rho(P)$.
2. **if** $\text{PARENT}(P) \neq \text{NULL}$ (i.e., $P \neq \text{root}$) **then**
 - Acquire $\epsilon(\text{Count})$.
 - Decrement Count by one.
 - **if** $(\text{Count} = 0)$ **then** release $\rho(\text{PARENT}(P))$.
 - Release $\epsilon(\text{Count})$.
3. **if** P is a non-leaf node **then**
 - Let $\{E_i = (\text{REGION}(P_i), P_i) \mid i = 1, \dots, k\}$ be the entries in P such that $\text{REGION}(P_i) \cap W \neq \phi$.

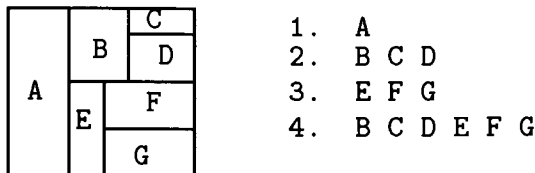


Figure 4.11: Joinable sets of rectangles.

- if ($k > 0$) then
 - $MyCount := k$.
 - For all entries E_i found above, continue the search in parallel invoking $Kt.Search(W \cap REGION(P_i), P_i, MyCount)$.
- else release $\rho(P)$.

4. if P is a leaf node⁵ then

- For all the points inside W return their ids.
- Release $\rho(P)$. □

4.3.2 Insert

As with the insert operation on the B^+ -Tree, we carry out an insert operation on the K-D-B tree in two phases. During the first phase, we locate the leaf node to which to add the new object and put the object there. The path in the tree we take is called the **insertion path**. During the downward traversal, we ω -lock a node and push it onto a stack if its child, which is also on the insertion path, is full. Otherwise, we empty the stack and release the ω -locks. At the end of the first phase, the entry representing a new point object is added to a leaf node. In phase two, if the stack is non-empty, we have to reorganize the tree.

In order to avoid the *up-and-down* splitting of nodes, we choose one split line for all nodes on the insertion path. When a leaf node, L , is to be split, we calculate the split line for L first. Let us assume that the split line is S_L . S_L is then used to split all the full nodes on the insertion path. Descendants of the split full nodes which intersect with S_L will also be split. We call these splits *forced splits*, because the nodes are being forced to split.

⁵I.e. a point node.

In Figure 4.12, the nodes A , B , C , and D form the insertion path and B is the first full node on it, i.e., C and D are both full nodes as well. The split line is initially applied to all children of B (B_1, \dots, B_k). Any child of B intersecting with S_L will be split. As D is a child of C , S_L will intersect with C , which implies that B will have at least one child (i.e., C) to be split. During the splittings of the children of B , B is allowed to temporarily overflow. S_L is applied to the children of B_i 's recursively down to the leaf nodes. If a descendant node P of B intersects with S_L and it has n entries, we split P into P' and P'' . After the split, there will be at most $2n$ new entries (if each old entry in P is divided into 2 entries). The entries that are not divided will be totally in either P' or P'' . The new entries are distributed between P' and P'' depending on their locations with respect to S_L . Therefore, neither P' nor P'' will have more than n entries. Consequently, none of the new nodes after splittings will overflow and a descendant node of B will be split at most once. Once the splittings have finished, updates will be propagated backward up to B . At B , B is split and its parent A is updated. The *forced splittings* simplify the procedures for the updating, but it has the same problem as discussed in [Ben75], where good utilization of storage cannot be guaranteed.

For an insert operation, the procedure *Kt.Select* is invoked to locate the leaf node to which add the new object, and the procedure *Kt.Adjust* is used to perform the *forced splitting*. More formally, *Insert*(O, R) is carried out as follows, where O is the point object, and R is the root of the tree. The new entry, $E_O = (POINT(O), O)$, is added to the leaf node found by procedure *Kt.Select*.

Kt.Insert(O, R)

1. $\pi := \Lambda$ (empty string).
2. $L := Kt.Select(O, R, NULL, \pi)$, /* This procedure returns an ϵ -locked leaf node, L , in which to place object O . π is changed by call-by-reference and, on return, gives a stack of full nodes, which forms a subpath of the insertion path. */
3. Add entry E_O to node L .
4. Downgrade $\epsilon(L)$ to $\omega(L)$.
5. **if** $FULL(L)$ **then**
 - Calculate the split line, S_L , for L .

- Let $Fnode$ be the first node in π .
- $Kt.Adjust(Fnode, S_L, \pi)$. □

In the following procedure, we define $REGION(NULL) = \phi$. Note that, when $Kt.Select(O, P, Parent, \pi)$ is recursively called, $Parent$ is ϵ -locked, and on exit, P is ω -locked and either $Parent$ is placed on π or $\epsilon(Parent)$ has been released.

$Kt.Select(O, P, Parent, \pi)$

1. Acquire $\epsilon(P)$.
2. **if** $Parent \neq NULL$ **then**
 - **if** $\neg FULL(P)$ **then**
 - Release all locks on nodes in π .
 - $FREE(\pi)$.
 - else**
 - Downgrade $\epsilon(Parent)$ to $\omega(Parent)$.
 - $PUSH(Parent, \pi)$.
3. **if** P is a non-leaf node **then**
 - Find an entry $E_s = (REGION(P_s), P_s)$ in P , where $POINT(O)$ is inside $REGION(P_s)$.
 - $Kt.Select(O, P_s, P, \pi)$.
- else**
 - Downgrade $\epsilon(Parent)$ to $\omega(Parent)$.
 - $PUSH(P, \pi)$.
 - Return P . □

When the following procedure is called, unless $L = NULL$, L is ω -locked.

$Kt.Adjust(L, S_L, \pi)$

1. **if** $(L = NULL)$ **then** return.
2. **if** $(L = Root)$ and $FULL(L)$ **then**

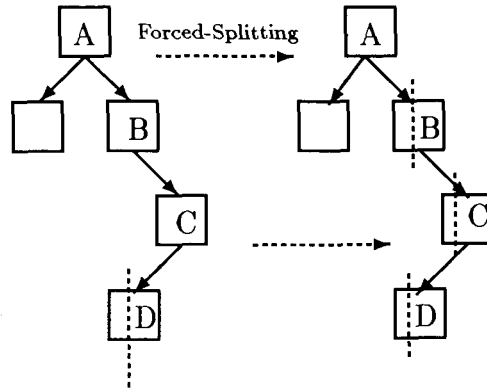


Figure 4.12: Forced splits.

- Acquire $\epsilon(L)$.
- Create a new root.
- Make it *Parent*.
- Add L to *Parent*.
- Acquire $\omega(\textit{Parent})$.
- Downgrade $\epsilon(L)$ to $\omega(L)$.

else

- $\textit{Parent} := L$.
- $L := \textit{FOLLOW}(P, \pi)$.

3. $\textit{Kt.Split}(L, S_L, \textit{Parent}, \pi)$.

4. Release all ω -locks for nodes on π .

5. Empty π (i.e., $\pi := \Lambda$). □

When the following procedure is called, unless $L = \textit{NULL}$, L is ω -locked, and except when $\textit{PARENT}(L) = \textit{NULL}$ (i.e., $L = \textit{root}$) or $\pi = \Lambda$, $\textit{PARENT}(L)$ is ω -locked.

$\textit{Kt.Split}(L, S_L, \textit{Parent}, \pi)$

1. **if** (L is not a leaf node) **then**

- Let L have k children, L_1, \dots, L_k , initially.

- For each $i = 1, \dots, k$, if L_i intersects S_L ,
 - Acquire $\omega(L_i)$.
 - $Kt.Split(L_i, S_L, L, \pi)$
- 2. Acquire $\epsilon(Parent)$.
- 3. Acquire $\epsilon(L)$.
- 4. Split L into L and L' with S_L .⁶
- 5. Add L' to $Parent$.
- 6. Update $Parent$.
- 7. Downgrade $\epsilon(L)$ to $\omega(L)$.
- 8. Downgrade $\epsilon(Parent)$ to $\omega(Parent)$.
- 9. **if** [L is not in π] **then** release $\omega(L)$.
- 10. **if** [$Parent$ is not in π] **then** release $\omega(Parent)$. □

4.3.3 Delete

We use two phases to carry out a delete operation. The first phase is to locate the leaf node which points to the object to be deleted. This phase is almost identical to $Kt.Select$, except that it also looks for possible underflow in the nodes on the **deletion path**. At the end of the first phase, the object is deleted.

In the second phase, we backtrack to reorganize the K-D-B tree as needed. The approach is either merging the entries of an underflowed node with its siblings or re-distributing entries from the siblings to the underflowed node. As noted previously, rectangular regions cannot be joined freely. The union of the regions will need to form a rectangle. This means an underflowed node may not be able to distribute its children to its siblings at all times.

Let R, P_1, \dots, P_k be the child nodes of the node P . R may have several *joinable sets*. We call the joinable set with respect to R containing the minimum number of nodes the **minimal joinable set (MJS)** of R . In the first approach, if R has underflowed, we first find its MJS. If the total number of entries in the nodes in the MJS is less than the maximum allowed (M), we can merge the nodes in the MJS together to form a new node, and remove the old nodes in the MJS. If the merging of entries is not possible for R , then the second

⁶ L may have more than k children after the splits above.

approach is to move some of the children in P_i 's to R . One method is to concatenate the P_i 's with R and recursively split the merged node until there is no overflow. Because of the non-overlapping property in the K-D-B tree, there may be many internal splits and up-and-down updates. To reduce the work, we first use P_i 's to calculate a split line (S_L) which does not intersect with R . S_L is then applied to split the children of the P_i 's. All children that are on the same side as R with respect to S_L become the children of R . However, during the calculation of S_L , we need to make sure that no more than M children will re-distribute to R . If R overflows, it will be split afterwards. With this method, there will be at most two downward propagations of splittings from R to the leaf nodes.

More formally, $Delete(O, R)$ under the lock-coupling method is carried out as follows.

Kt.Delete(O, R)

1. $\pi := \Lambda$ (empty string).
2. $L := Kt.Find(O, R, NULL, \pi)$, /* This procedure returns a ϵ -locked leaf node, L , in which object O is referenced. π is changed by call-by-reference and, on return, gives a stack of nodes, which forms a subpath of the deletion path. */
3. Remove entry E_O in node L .
4. Downgrade $\epsilon(L)$ to $\omega(L)$.
5. Let L be the first under-utilized node in π .
6. *Kt.Condense*(L, π).
7. Release all locks in π and empty π . □

In the following procedure, we define $REGION(NULL) = \phi$. Note that, when *Kt.Find*($O, P, Parent, \pi$) is recursively called, $Parent$ is ϵ -locked, and on exit, P is ω -locked. For $Parent$, it is either placed on π with $\omega(Parent)$ or its ϵ -lock has been released.

Kt.Find($O, P, Parent, \pi$)

1. Acquire $\epsilon(P)$.
2. **if** $Parent \neq NULL$ **then**
 - **if not** ($MIN(P)$ or $FULL(P)$) **then**
 - Release all locks on nodes in π .

– $FREE(\pi)$.

else

– Downgrade $\epsilon(Parent)$ to $\omega(Parent)$.

– $PUSH(Parent, \pi)$.

3. if P is a non-leaf node then

- Find an entry E_s in P , where O is inside $REGION(P_s)$.
- $Kt.Find(O, P_s, P, \pi)$.

else

- **if there is an entry E_s in P then return P .**
- **else return $NULL$.**

□

When the following procedure is called, unless $U = NULL$, U is ω -locked, and except when $PARENT(U) = NULL$ (i.e., $U = Root$) or $\pi = \Lambda$, $PARENT(U)$ is ω -locked.

$Kt.Condense(U, \pi)$

1. if $U = NULL$ then return.

2. if U has no child then

- Acquire $\epsilon(U)$.
- Mark U as deleted.
- Release $\epsilon(U)$.
- Return.

3. if U is a leaf node then return.

4. if $U = Root$ then

- $Condense(FOLLOW(U, \pi), \pi)$.
- Return.

5. $Parent := PARENT(U)$.

6. Let $Parent$ has k children, P_1, \dots, P_k , where $P_m = U$.

7. Acquire $\omega(P_i)$ for $i = 1, \dots, k$.

8. Let J be the MJS w.r.t. U .
9. **if** number of entries in $J < Max_{entry}$ **then**
 - Acquire $\epsilon(Parent)$.
 - Acquire $\epsilon(P_i)$ for $i = 1, \dots, k$.
 - Move all the entries of J to U .
 - Update $Parent$.
 - Downgrade $\epsilon(Parent)$ to $\omega(Parent)$.
 - Release ϵ -locks of the children of $Parent$ except U .
 - Downgrade $\epsilon(U)$ to $\omega(U)$.
 - Let L be the first under-utilized node in π .
 - $Condense(L, \pi)$.

else

- Let S_L be a split line for J which does intersect with U .
- Release ω -locks of the children of $Parent$ except U .
- $Kt.DSsplit(\pi, U, S_L)$.
- $Condense(U, \pi)$. □

$Kt.DSsplit(\pi, U, S_L)$

1. **if** $(\pi = NULL)$ or $(U = NULL)$ **then** return.
2. Let L be the *first full node* in π .
3. **if** $(L = Root)$ **then**
 - Create a new root.
 - Make it $Parent$.
 - Acquire $\epsilon(Parent)$.
 - Add L to $Parent$.
- else** $Parent := PARENT(L)$.
4. Let L has k children, L_1, \dots, L_k .
5. For each L_i where $i=1, \dots, k$ and L_i intersects S_L .

- Acquire $\omega(L_i)$.
 - $Kt.Split(L_i, S_L, NULL)$.
 - Downgrade $\epsilon(L_i)$ to $\omega(L_i)$.
6. Acquire $\epsilon(Parent)$.
 7. Acquire $\epsilon(L)$.
 8. Now L has k' children, L'_1, \dots, L'_k .
 9. Split L into L and L' with S_L .
 10. Add L' to $Parent$.
 11. Update $Parent$.
 12. Release all locks except for nodes in π .
 13. Remove all nodes in π from L to $PARENT(U)$. □

4.3.4 Correctness of the Operations

The structure of a K-D-B tree is similar to the B^+ -tree discussed in the Section 4.1. The major differences between them are the ways the updating process due to splitting and condensation of nodes are done. Most of the correctness proofs for our algorithms for a K-D-B tree are the same as those for a B^+ -tree. Therefore, we will only discuss the issues related to reorganization in detail. In the following discussion, an *update operation* means either an insert or a delete operation.

Lemma 4.5 *A search operation cannot overtake an update operation if they are traversing the same path. Similarly, an update operation cannot overtake any other operation.*

Proof: The proof is the same as in Lemma 4.1 in the B^+ -tree. □

Theorem 4.5 *Protocol KT is both deadlock and livelock free.*

Proof: The proof is the same as in Theorem 4.1. □

Lemma 4.6 *In the protocol KT, no search operation will miss an available object.*

Proof: We first consider the case where a search operation is traversing the same path together with an update operation in its first phase. Let U_1 be an update operation and S_1 be a search operation. Similarly to the B^+ -tree, during the downward traversal of U_1 , it does not modify any internal nodes except a leaf node, say L . Hence, S_1 will not miss any node it should visit and always arrive at L correctly. If S_1 ρ -locks L first, then U_1 will need to wait. On the other hand, S_1 will wait when U_1 ρ -locks L first. In either cases, S_1 will see the available objects in L correctly.

Suppose there is a cleanup or a condense operation backtracking along the same path that S_1 uses. Let S_1 visit node P . It will first acquire a ρ -lock on P and then determine the nodes to visit next. Let P_c be one of the nodes in P_1, \dots, P_k to be visited by S_1 next. If U_1 is in its second phase, trying to modify P_c , then P_c can only be changed if U_1 has acquired an $\epsilon(P_c)$. If this happens, S_1 will wait until U_1 modifies and releases the lock. S_1 still correctly arrives at P_c because P_c cannot be split or removed, since S_1 holds a ρ -lock on P . U_1 might have changed the content (e.g., the number of entries) in P_c . However, this does not affect the results of S_1 because U_1 must have finished its work at the corresponding leaf node already. If S_1 gets a ρ -lock on P_c before U_1 locks it, then S_1 will work on P_c as intended and U_1 will wait until S_1 finishes. Hence, the search operation will not miss any object which is available. \square

Theorem 4.6 *The protocol KT is correct.*

Proof: From Lemma 4.5, Lemma 4.6, and Theorem 4.5, we can conclude that the protocol KT are correct. \square

4.4 Quad-B Tree

All the previous index structures described in this chapter have some serious disadvantages. For example, with the K-D-B tree, there is a problem of balancing. Furthermore, the locks for many nodes must be hold and up-and-down splitting of nodes also occur frequently. The **quad-B tree** presented below overcomes these problems to a certain degree; it was motivated by the work in [NgK93, Ng94b, NgK95].

The quad-B tree has two distinguishing properties which affect the concurrency control algorithms for it.

- The keys within the tree are scalar values which are totally ordered.

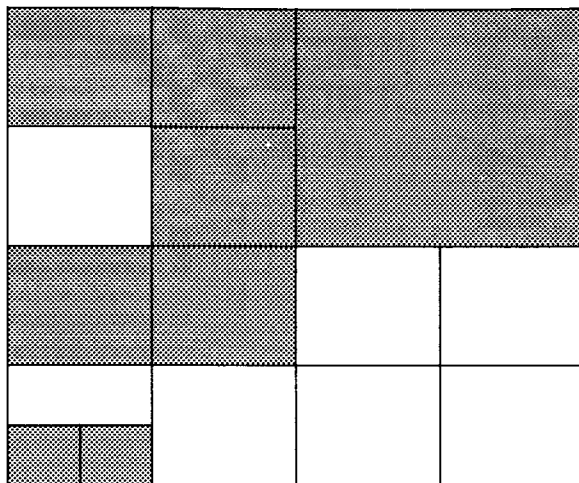


Figure 4.13: The shaded regions contain one or more point objects.

- The MBRs associated with the internal nodes are allowed to overlap. This makes the balancing of a tree easier and avoids excessive partitioning of the data space.

For many years, **quadtrees** have been used as the index structure for spatial objects such as lines, points, and rectangles. A quadtree [Sam80] presents a region obtained by a recursive 4-way subdivisions of the data space into quadrants. If a resulting quadrant contains more than a certain number of points, then it is further subdivided. At depth i , the height (width) of a quadrant is 2^{-i} times the original height (width) of the data space. In Figure 4.13, the regions containing data objects are shaded. The corresponding quadtree is shown in Figure 2.3. In [Sam88], Samet describes several variants of quadtree, such as the *point quadtree*, the *MX quadtree* and the *PR quadtree*. In general, there are three methods to represent the regions resulting from the 4-way subdivisions. The first method, the *transformation method* [KLR79], encodes the size and coordinate information of each region. This method enables the use of an ordered list of keys to represent the regions. The second method [Sam82, HuS79, DRS80] uses some kind of tree. This method is a natural representation of a 4-way subdivisions but the tree may not be balanced. The third method, as described in [Jol84], uses a hybrid scheme where a region is represented by a set of segments and each segment corresponds to a numeric key. The third method constructs a forest of quadtrees, which makes concurrency control algorithms harder to achieve. Note that the second method allows the regions to be represented by a B^+ -tree, which is balanced.

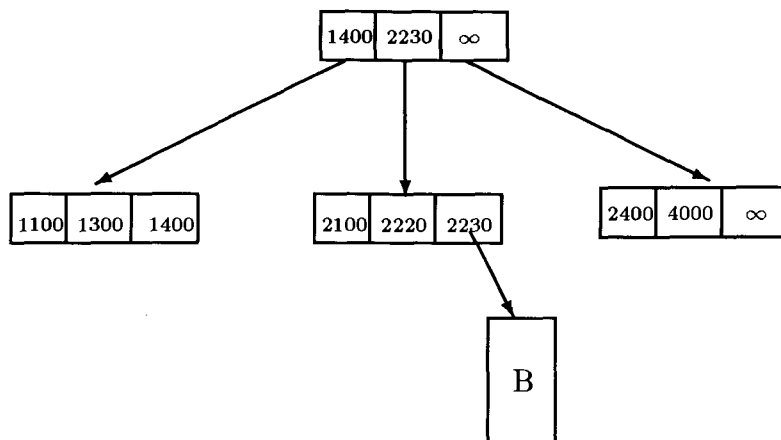


Figure 4.14: A Quad-B tree.

The quad-B tree we propose is based on the idea described in [Abe84]. We call a non-empty quadrant which does not contain a smaller quadrant a **terminal quadrant**. We represent each terminal quadrant by a scalar value and store it at the leaf level of a B^+ -tree. With the scalar transformation on their sizes and locations in the data space, the terminal quadrants can be linearly ordered. We can then use the conventional concurrency algorithms for the B^+ -trees. To support spatial queries better, we also add the MBR information to every node in the B^+ -tree.

More formally, a quad-B tree is a representation of the terminal quadrants for point data. It is a B^+ -tree such that each non-leaf node contains entries for its children and a MBR contains all regions covered by the children. Each entry is of the form $(MBR, Value, P)$. MBR is the MBR associated with a child node P^7 and $Value$ is a scalar value which is the largest scalar values amongst the entries in P . At each level, the rightmost entry of the rightmost node has the largest possible value in the subtree. For the internal nodes, MBR's are allowed to overlap. A leaf node contains entries of the form $(MBR, Value, B)$, where B is a bucket of point objects, and MBR is the minimum quadrant to cover all the points in B . $Value$ is the corresponding scalar value for MBR . The maximum number of points that can be stored in B is M . All entries in the leaf nodes represent different terminal quadrants, and, therefore there are no overlap among them.

We represent each quadrant by a scalar value obtained from an encoding scheme. In

⁷To be precise, P is a pointer to a child node. We often identify the node and a pointer to it.

our encoding scheme, we use the approach described in [Abe84]. A quadrant at depth d (where $d > 0$) is encoded as follows. Let D be the largest depth of any terminal quadrant. Suppose the entire data space is encoded as 0. Let the quadrant q have the key value k and its parent q_p have the key value k' . Then

$$k = k' + s5^{D-d} \quad (4.1)$$

where

$$\begin{aligned} s &= 1, \text{ if } q \text{ is the northwest son of } q_p \\ s &= 2, \text{ if } q \text{ is the southwest son of } q_p \\ s &= 3, \text{ if } q \text{ is the southeast son of } q_p \\ s &= 4, \text{ if } q \text{ is the northeast son of } q_p \end{aligned}$$

Throughout the section, we will represent the key values in base 5 and the above transformation by $\varphi(R)$, where R is a quadrant. In Figure 4.15, we show the encoded values for the shaded regions in Figure 4.13 and the same values are stored at the leaf nodes shown in Figure 4.14.

We now discuss how to support concurrent operations in a quad-B tree. The *lock-coupling* protocol, which was used in the previous point index structures, is first used. We will also use the *link technique* described in [LeY81] together with the lock-coupling approach. If it is done, a search operation is allowed to release the lock on a node it holds before it obtains a lock on the next node. It offers more concurrency than the lock-coupling alone, but requires the addition of new edges (*link pointers*) to the tree in order to avoid anomalies.

When a node overflows, it will be split. However, splitting is handled differently for leaf nodes and internal nodes. At a leaf node, when one of its entries has in its bucket more than M point objects, the entry will be replaced by the new entries corresponding to its quadrants. Only the quadrants which contain data are transformed to key values. They are then added to the leaf node. For an internal node, a node is split when it has more than M entries and it is replaced by two new nodes.

We introduce a new operation on a quad-B tree, in addition to the first three operations discussed in Chapter 3. The new operation is also a *search* operation but it tries to find an object at a given location (i.e., point search operation). This operation traverses only one search path from the root to a leaf node. The operation is stated below.

$P\text{Search}(P, R)$ Search if there is any object at point P in the tree rooted at R

1100	1400	4000
	1300	
2100	2400	
	2200	

Figure 4.15: Scaler values for the terminal quadrants.

4.4.1 Lock-Coupling

In this section, the quad-B tree is adopted with the lock-coupling approach. Our concurrency control protocol, QB, for quad-B trees is implemented by a collection of procedures, for search, insert, and delete operations, as well as those for tree-reorganization. In the following sections, we discuss them in detail.

Search

In this section, we want to discuss two types of search operations: the *window* search and *point* search operations. A window search operation tries to find all the point objects within a given window; while a point search operation tries to locate if there is any object at a specific point.

A search operation starts at the root of a tree and descends down to the leaf level. At each visited node, a *window* search operation may branch into multiple sub-search operations. For the same reason as for the search operations in the B⁺-tree and R-tree, we use a lockable variable, *Count*, in the window search operation.

More formally, $Search(W, R)$ is carried out by calling $Qb.RSearch(W, R, Count)$ given below, after setting $Count := 0$. *Count* in the following procedure is a call-by-reference parameter.

$Qb.RSearch(W, R, Count)$

1. Acquire $\rho(R)$.
2. **if** $PARENT(R) \neq NULL$ (i.e., $R \neq root$) **then**
 - Acquire $\epsilon(Count)$.
 - Decrement $Count$ by one.
 - **if** $(Count = 0)$ **then** release $\rho(PARENT(R))$.
 - Release $\epsilon(Count)$.
3. **if** R is a non-leaf node **then**
 - Let $\{E_i = (MBR(R_i), R_i, V_i) \mid i = 1, \dots, k\}$ be the entries in R such that $MBR(R_i) \cap W \neq \phi$.
 - **if** $(k > 0)$ **then**
 - $MyCount := k$.
 - For all entries E_i found above, continue the search in parallel invoking $Qb.RSearch(W \cap MBR(R_i), R_i, MyCount)$.
 - else** release $\rho(R)$.
4. **if** R is a leaf node **then**
 - For all points inside W return their object ids.
 - Release $\rho(R)$. □

A *point* search operation works similarly to a search operation in a B-tree. There is a unique path for it to traverse before reaching a leaf node which may contain the object. Similarly to the *window* search operations, a *point* search operation starts at the root of a tree and descends down to the leaf level. However, at each visited node, the search operation compares the key values in the node against the scalar value representing the $POINT(P)$. Only one child will be visited next and the $Count$ variable is not needed. More formally, $PSearch(P, R)$ is carried out by calling $Qb.PSearch(P, SValue, R)$ given below, where P is a point, $SValue$ is the scalar value for P after transformation, and R is the root of the tree. The operation is similar to the search operation in a B^+ -tree. Therefore, the details of the operation is omitted here.

Insert

Similarly to the R-tree, there are two phases for an insert operation. During the first phase, we locate the leaf node to which to add the new object. The path in the tree we take is called the **insertion path**. After adding a new object to a leaf node (i.e., in the second phase), we may have to perform some maintenance operations by tracing back the insertion path to

1. reorganization of the tree, and/or
2. enlarge the MBR's of some nodes.

The definition of an *unsafe* node is the same as that for the R-tree. ω -locks are used to avoid multiple updating operations working in the same scope. The only difference is at leaf nodes. Let M be the maximum number of entries allowed at a leaf node. If a leaf node contains $M - 3$ or more entries, then there is a possibility that it may be split during the second phase. If a split does occur, the parent of the node may then have from one to four new entries. More formally, $Insert(O, R)$ under the lock-coupling method for the quad-B tree is carried out as follows.

Qb.Insert(O, R)

1. $\pi := \Lambda$ (empty string).
2. $SValue := \varphi(O)$.
3. $L := Qb.Select(O, SValue, R, NULL, \pi)$, /* This procedure returns an ϵ -locked leaf node, L , in which to place point O . π is changed by call-by-reference and, on return, gives a stack of unsafe nodes, which forms a subpath of the insertion path. */
4. **if** there is an entry E_O whose region contains O **then**
 - Let B be the data bucket referenced by E_O .
 - Add O to B .
 - **if** B is full **then**
 - Let A be the region corresponding to E_O .
 - Split A into quadrant with at most 4 non-empty quadrants as A_1, A_2, A_3, A_4 .
 - Distribute objects in B to B_i 's for the corresponding A_i 's.

– Let E_i 's be the new entries to reference the B_i 's.

else

- Create an entry E_i which is a neighborhood quadrant of E_O .
- Let B be the data bucket referenced by E_i .
- Add O to B .

5. **if** E_i 's exist **then** add entries E_i 's in node L .

6. Downgrade $\epsilon(L)$ to $\omega(L)$.

7. $Qb.CleanUp(L, \pi)$. □

In the following procedure, we define $MBR(NULL) = \phi$. Note that, when $Qb.Select(O, SValue, P, Parent, Path)$ is recursively called, $Parent$ is ϵ -locked, and on exit, P is ω -locked. For $Parent$, it is either placed on $Path$ with an ω -lock or $\epsilon(Parent)$ has been released.

$Qb.Select(O, SValue, P, Parent, Path)$

1. Acquire $\epsilon(P)$.

2. **if** $Parent \neq NULL$ **then**

- $unsafe := 0$.
- **if** P is a leaf node **then**
 - **if** $Size^8(P) > Max_{allow} - 3$ **then** $unsafe := 1$.

else

- **if** $Size(P) = Max_{allow}$ **then** $unsafe := 1$.
- **if** O outside $MBR(Parent)$ **then**
 - $unsafe := unsafe + 2$.
 - Update $MBR(Parent)$ with O .
- **if** $unsafe \geq 1$ **then**
 - Downgrade $\epsilon(Parent)$ to $\omega(Parent)$.

⁸ $Size$ is a function which returns the number of entries in a node.

– $PUSH(Parent, Path)$.

- **if** ($unsafe = 0$) **then** release ω -lock on $Parent$.
- **if** ($(unsafe = 0)$ and $(Path \neq \Lambda)$) **then** release all locks on nodes in $Path$, and $FREE(Path)$.

3. **if** P is a non-leaf node **then**

- Find an entry E_s in P , where $V_{s-1} < SValue \leq V_s$.
- $Qb.Select(O, SValue, P_s, P, Path)$.

else return P . □

When the following procedure is called, unless $L = NULL$, L is ω -locked, and except when $Parent = NULL$ (i.e., $P = root$) or $Path = \Lambda$, $PARENT(L)$ is ω -locked.

$Qb.CleanUp(L, Path)$

1. **if** $L = NULL$ **then** return.

2. **if** $OVERFL(L)$ **then**

- **if** $PARENT(L)$ exists **then**
 - Upgrade $\omega(PARENT(L))$ to $\epsilon(PARENT(L))$.

else

- Create a new root.
- ϵ -lock the new root.
- Make it $PARENT(L)$.
- Acquire $\epsilon(PARENT(L))$.
- Acquire $\epsilon(L)$.
- Split L into L and L' .
- Add L' to $PARENT(L)$.
- Update $PARENT(L)$.
- Downgrade $\epsilon(PARENT(L))$ to $\omega(PARENT(L))$.
- Release $\epsilon(L)$.
- $L := POP(Path)$.

- $Qb.CleanUp(L, Path)$.

else

- Release all locks on $Path$.
- $FREE(Path)$. □

Delete

We use two phases to carry out a delete operation. The first phase is to locate the object. This phase is almost identical to a *point* search operation. Internal nodes are ρ -locked and the bounding rectangles are not shrunk on its way down. Similarly to the *insert* operations, it upgrades the ρ -locks to ω -locks when the nodes are considered *unsafe*. For a deletion, a node is unsafe if the deletion may cause

1. reorganization of the tree, and/or
2. shrinkage of the MBR's of some nodes.

The first situation may happen when a node has the minimum number of entries. The second situation arises when the deletion of the point object results in a change of the MBR. This happens at a node if the removal of the next visited node will shrink its MBR. At the end of phase one, if the object has been found, it is removed and the tree is reorganized in the second phase.

In the second phase, we backtrack the tree along the path of nodes which are ω -locked. More formally, $Delete(O, R)$ under the lock-coupling method is carried out as $Qb.Delete(O, R)$ where O is the object to be deleted and R is the root of the tree initially. Procedure $Qb.Find$ is invoked to perform the first phase of a deletion and ω -lock the unsafe nodes. The second phase is carried out by the procedure $Qb.Condense$.

$Qb.Delete(O, R)$

1. $\pi := \Lambda$ (empty string).
2. $SValue := \varphi(O)$.
3. $L := Qb.Find(O, SValue, R, NULL, \pi)$, /* This procedure returns a ϵ -locked leaf node, L , in which object O is referenced. π is changed by call-by-reference and, on return, gives a stack of under-utilized nodes, which forms a subpath of the deletion path. */

4. Let E_O be an entry in L whose region contains O .
5. **if** ($E_O = NULL$) **then** return.
6. Let B be the data bucket referenced by E_O .
7. Remove O from B .
8. **if** B is empty **then**
 - Remove entry E_O in node L .
9. Try to merge entries in L by joining different regions of entries.
10. Downgrade $\epsilon(L)$ to $\omega(L)$.
11. $Qb.Condense(L, \pi)$. □

In the following procedure, we define $MBR(NULL) = \phi$. Note that, when $Qb.Find(O, SValue, P, Parent, Path)$ is recursively called, $Parent$ is ϵ -locked, and on exit, P is ω -locked. $Parent$ is either placed on $Path$ with an ω -lock or $\epsilon(Parent)$ has been released.

$Qb.Find(O, SValue, P, Parent, Path)$

1. Acquire $\epsilon(P)$.
2. **if** $Parent \neq NULL$ **then**
 - $unsafe := 0$.
 - **if** $UNDERFL(P)$ **then** $unsafe := 1$.
 - **if** P is the only child in $Parent$ and $POINT(O)$ is inside $MBR(P)$ **then**
 - $unsafe := unsafe + 2$.
 - **if** $unsafe \geq 1$ **then**
 - Downgrade $\epsilon(Parent)$ to $\omega(Parent)$.
 - $PUSH(Parent, Path)$.
 - **if** ($unsafe = 0$) **then** release ϵ -lock on $Parent$.
 - **if** ($(unsafe = 0)$ and $(Path \neq \Lambda)$) **then** release all locks on nodes in $Path$, and $FREE(Path)$.
3. **if** P is a non-leaf node **then**

- Find an entry E_s in P , where $V_{s-1} < SValue \leq V_s$.
- $Qb.Find(O, SValue, P_s, P, Path)$.

else return P . □

When the following procedure is called, unless $L = NULL$, L is ω -locked, and except when $PARENT(P) = NULL$ (i.e., $P = root$) or $Path = \Lambda$, $PARENT(L)$ is ω -locked.

$Qb.Condense(L, Path)$

1. if $(L = NULL)$ or $((\neg UNDERFL(L))$ and $(Path = \Lambda))$ then

- Release $\omega(L)$.
- Return.

2. $Parent := POP(Path)$.

3. if $(Parent = NULL)$ and $(L$ is empty) then

- Acquire $\epsilon(L)$.
- Mark L as deleted if L has no child.
- Set root of the tree to $NULL$.
- Release $\epsilon(L)$.
- Return.

4. Acquire $\epsilon(Parent)$.

5. if $UNDERFL(L)$ then

- Let $Parent$ has k children as P_1, \dots, P_k including L .
- Acquire $\epsilon(P_i)$ for $i = 1, \dots, k$.
- Merge the entries of P with its siblings.
- Update $Parent$ and its MBR .
- Update the MBR 's of P_i 's.
- Remove L .
- Release $\epsilon(P_i)$ for $i = 1, \dots, k$.

else

- Update *Parent* and its *MBR*.
 - Release $\omega(L)$.
6. Downgrade $\epsilon(\textit{Parent})$ to $\omega(\textit{Parent})$.
 7. *Qb.Condense(Parent, Path)*. □

Correctness Proof

The proofs of correctness of the operations are the same as those for the B^+ -tree. Therefore, we will only state a lemma and two theorems without proof.

Lemma 4.7 *A search operation cannot overtake an update operation if they are traversing the same path. Similarly, an update operation cannot overtake any other operation.*

Theorem 4.7 *The protocol QB is both deadlock and livelock free.*

Theorem 4.8 *The protocol QB based on the lock-coupling protocol is correct.*

4.4.2 Link Extension

The link technique was adapted by Lehman and Yao to B-trees [LeY81]. They modified the B-tree to the **B-link tree** by adding a link pointer to each node. Each node of a B-link tree thus has two types of pointers: its **child pointers** point to the child nodes, while its **link pointer** points to its right neighbor at the same level of the tree. Therefore, link pointers provide an alternate path from a node to each of its child nodes via the leftmost child node. We take advantage of this redundancy to support concurrent operations in a quad-B tree.

With the link extension, in a quad-B tree, there is a link pointer besides the child pointers for each node as shown in Figure 4.16 by dashed arrows. At each level of the tree, there is a head link pointer pointing to the leftmost node of the level. Thus, every node in the tree can be accessed either via the child-parent pointers or the link pointers. The updating operations, such as insertions and deletions, use the lock-coupling protocol as before. However, the search operations utilize the link pointers to enhance the level of concurrency.

Before visiting a new node, in the link extension, a search operation releases the lock on the previous node. This allows overtaking by other operations. For example, an insert operation may overtake and make changes to some nodes before a search operation arrives

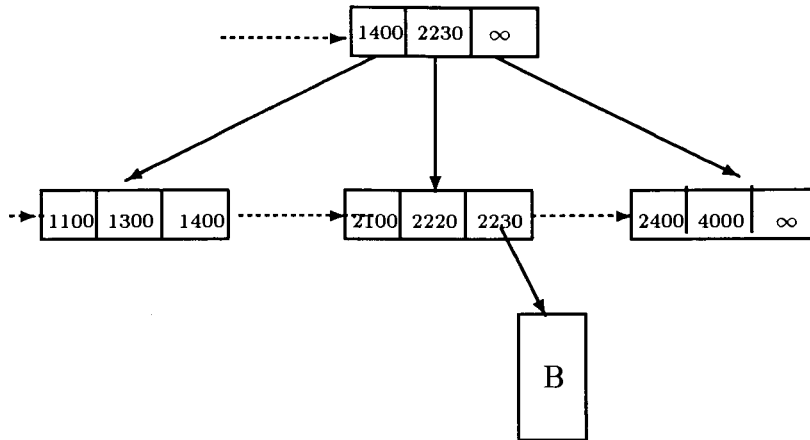


Figure 4.16: A Quad-B tree with link extension.

	ρ	χ	ω	ω_d	ϵ
ρ	1	1	1	1	0
χ	1	1	1	0	0
ω	1	1	0	0	0
ω_d	1	0	0	0	0
ϵ	0	0	0	0	0

Figure 4.17: Lock compatibility matrix for a quad-B tree with link technique.

at the next visited node (which can be a leaf node). When this happens, serializability may be violated. To solve this problem, we use two methods. The first method is to include a timestamp for each point object at the leaf nodes. The timestamp of an object records the time when the object is inserted into the quad-B tree. Using the timestamps, search operations are restricted to see only those objects which are inserted by older insert operations. The second method introduces two additional types of locks to avoid overtaking between delete and search operations. Besides the three lock types used in Chapter 3, we introduce the χ -lock and the ω_d -lock. A χ -lock is used by a search operation to prevent any delete operation from entering its scope and an ω_d -lock is used by a delete operation to prevent any other operations from overtaking it. The new lock compatibility matrix is shown in Figure 4.17.

The concurrency control protocol, QBL, for quad-B trees is adopting the link technique.

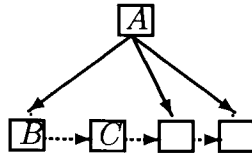


Figure 4.18: Using the link pointer to search.

It is implemented by a collection of procedures, for search, insert, and delete operations, as well as those for tree-reorganization. In the following sections, we discuss them in detail.

Search

Like the search operations in the previous section, a search operation starts at the root of the tree and descends down to the leaf level. However, unlike the former operations, each search operation here releases its lock before visiting other nodes in the tree. When a search operation, S_1 , visits node A in Figure 4.18, a ρ lock on it is first acquired. After it has decided which child nodes to visit next, the ρ lock is released. When it visits node B , B may have been split and the right sibling of B (i.e., C) is a new node whose corresponding entry has not been put in A . When this happens, the link pointer at B provides an alternate way to access node C . In our implementation, when a search visits B , it verifies the scalar value of B stored at A against the scalar value of the rightmost entry of B . If they are different, a split must have been occurred, and a new sub-search operation will be spawned off to search the subtree rooted at C .

A window search operation utilizes the MBRs of nodes to traverse the tree. However, the *lockable* variable, *Count*, is no longer required. Instead, we use the scalar values of the nodes to determine if rightward searches are needed because of splitting. If a window search operation traverses a single path downward, then an update operation (i.e., either an insert or a delete) will not cause any non-serializable execution. However, a problem may occur when a search is divided into several sub-searches. In this case, some update operations may overtake the sub-searches and reach the leaf nodes first to cause a non-serializable execution as observed in the B^+ -tree. In the lock-coupling approach, we can make overtaking impossible. However, because a search operation releases a lock on a node before acquiring a lock on the next visited node, overtaking by other update operations along the same path is possible. We now will consider insertion and deletion separately. In

this section, we present how to prevent overtaking between search and delete operations. The solution for the insert operation is discussed in a later section.

As the problem is due to multiple sub-searches of a search operation, a search may place a lock on the node it visits to prevent a delete operation from entering its scope. The lock should be compatible with most operations except the delete operations. We define a branch node of a search operation as follows.

Branch Node(BN): A node is a *branch node* of a search operation S if it is the first node visited by S where there are multiple entries in the node whose MBRs overlap with the search window. I.e., more than one sub-search starts after visiting the node for S .

Our solution is to place a χ -lock on the branch node visited by a window search operation. As shown in the lock compatibility matrix, the χ -lock is compatible with all other lock types except the one used by delete operations. This method is based on the assumption that deletions are infrequent; otherwise, they are easily blocked at higher levels of the tree. The window search operations are carried out by calling

$QbL.RSearch(W, R, OldValue, Bnode, TotSearch)$, where W is the search window, R is the root of the tree, $OldValue$ is the scalar value of R obtained from the parent of R , $Bnode$ is the branch node and $TotSearch$ is the number of sub-search operations. Initially, $OldValue$ is the value of the largest possible number, $Bnode$ is $NULL$ and $TotSearch$ is 0. In the algorithm, when a search operation divides into several sub-searches, it will place a χ -lock on $Bnode$. The counter, $TotSearch$, is increased (decreased) whenever sub-searches are created (finished). When the counter equals 0, the χ -lock on $Bnode$ is released if $Bnode$ is not $NULL$.

$QbL.RSearch(W, R, OldValue, Bnode, TotSearch)$

- Acquire $\chi(R)$.
- **if** R is a non-leaf node **then**
 - Let $\{E_i = (MBR(R_i), R_i, V_i) \mid i = 1, \dots, k\}$ be the entries in R such that $MBR(R_i) \cap W \neq \phi$.
 - **if** $(k > 0)$ **then**
 - * **if** $(Bnode = NULL)$ and $(k > 1)$ **then**
 - $Bnode := R$.

- * Acquire $\epsilon(TotSearch)$.
- * $TotSearch := TotSearch + k$.
- * Release $\epsilon(TotSearch)$.
- * For all entries E_i found above, continue the search in parallel invoking $QbL.RSearch(MBR(R_i) \cap W, R_i, V_i)$.

else for all points inside W return their object ids.

- Let R has n entries.
- **if** ($OldValue > V_n$) and ($NEXT(R) \neq NULL$) **then**
 - $R' := NEXT(R)$.
 - $QbL.RSearch(W, R', OldValue, Bnode, TotSearch)$.

else

- Acquire $\epsilon(TotSearch)$.
- $TotSearch := TotSearch - 1$.
- Release $\epsilon(TotSearch)$.
- **if** ($TotSearch = 0$) and ($Bnode \neq NULL$) **then** release $\chi(Bnode)$.
- **if** ($R \neq Bnode$) **then** release $\chi(R)$. □

Insert

In the link extension, when a node is split, a link pointer is created to *link* the two nodes first. The *link* is important because it routes other operations to the missing part of a split node. As illustrated in Figure 4.19, there are two steps in splitting. The first step establishes the link pointers and the second step updates the parent of the split nodes.

In the section on search operations, we discussed a method to avoid inconsistency between delete and search operations. If the same method is used for insertions, then insertions will be blocked easily just as deletions. Consider a scenario where S_1 is a search operation and I_1 is the insert operation adding object O . Suppose I_1 overtakes S_1 . We observe that the first phase of I_1 does not change the result of S_1 if O is ignored from the result of S_1 . This observation suggests us to use time-stamping to solve the problem. Each insert operation will have a timestamp which is the time when it accesses the root of the tree. Each

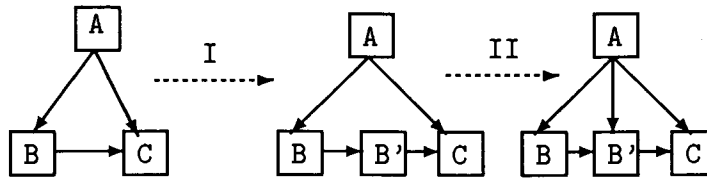


Figure 4.19: Two steps of a split.

object at a leaf node is timestamped with the timestamp of the operation which inserted the object. When a search operation reaches the leaf node, it compares its starting time with the timestamps of the objects. An object with a later timestamp is ignored. The result of this procedure makes sure search operations can not see objects added by the overtaking insertion. To implement the method, we would need to make a small change in *QbL.WSearch* accordingly.

The first phase of an *insert* operation is carried out by the procedure *QbL.Select* which locates the leaf node and expands the MBRs of some internal nodes. After the insertion of the object, if the leaf overflows, the *insert* operation will split it into quadrants. The second phase is then performed by invoking the procedure *QbL.CleanUp*. More formally, *Insert(O, R)* has two parameters. Initially, *O* is the object to be added, and *R* is the root of the tree.

QbL.Insert(O, R)

1. $SValue := \varphi(O)$.
2. $L := QbL.Select(O, SValue, R, Path)$.
3. **if** there is an entry E_O whose region contains O **then**
 - Let B be the data bucket referenced by E_O .
 - Add O to B .
 - Store the starting time of the operation together with O .
 - **if** B is full **then**
 - Let A be the region corresponding to E_O .
 - Split A into quads with at most 4 non-empty quads as A_1, A_2, A_3, A_4 .

- Split B into B_i 's for the corresponding A_i 's.
- Let E_i 's be the new entries to reference the B_i 's.

else

- Create entry E_i whose region is a neighborhood quadrant of E_O and O is inside the region.
 - Add O to B .
4. if E_i 's exist then add entries E_i 's in node L .
 5. Release $\epsilon(L)$.
 6. $QbL.CleanUp(L, SValue, O, Path)$. □

The supporting procedure $QbL.Select$ and $QbL.CleanUp$ are implemented in the same way as $Qb.Select$ and $Qb.CleanUp$, respectively. Therefore, they are skipped here.

Delete

As in the lock-coupling method, there are two phases to implement the *delete* operation. The first phase is to locate the leaf node L which contains the point object O and to delete it. The ω_d -locks are used to prevent undesirable overtaking. In the second phase, we perform the required cleanup using the “mark-and-remove” approach. If a node becomes empty as a result of deleting O from L , then it is flagged as “deleted.” The link pointer of the node will be changed and pointed to the head link pointer at the current level of the node. “Deleted” nodes are later garbage-collected periodically, because there can be search operations working on it.

There are two parameters for $QbL.Delete(O, R)$ where O is the object to be deleted and R is the root of the tree initially. The implementation of $QbL.Delete$ is the same as $Qb.Delete$ except ω_d -locks are used instead of ω -locks. Therefore, the details of the algorithm is skipped here.

4.4.3 Correctness of the Operations

In this section, we prove that the protocol QBL for the quad-B tree is correct.

Lemma 4.8 *A window search operation cannot overtake an update operation along the same access path.*

Proof: Consider a window search operation (S) and an update operation (U) in its first phase visit node P at the same time. If U acquires $\epsilon(P)$ first, because of lock-coupling, it will hold the lock until it acquires the ϵ -lock of the node next visited. U may then release the lock on P or downgrade the lock to an ω -lock or an ω_d -lock. In both cases, S visits the node after U has finished its work at P . Hence, a window search operation cannot overtake an update operation. \square

Lemma 4.9 *An insert operation can overtake a window search operation.*

Proof: Suppose a window search operation (S) and an insert operation (I) both visit node P . If S acquires its χ -lock on P first, I , which request an $\epsilon(P)$, will wait until the χ -lock is released. S will release the lock right after determining the next visiting node, say P_n . At the same time, I can acquire its lock on P , complete its work, and even overtakes S by placing a lock on P_n first. \square

Lemma 4.10 *Two update operations cannot overtake each other.*

Proof: Two update operations traversing along the same path cannot overtake each other because ϵ -lock-coupling is used for the insert and deletion operations. \square

Lemma 4.11 *A search operation will always find a node at each level of the tree to continue its downward traversal.*

Proof: Consider a search operation (S) visiting node P . If P is an internal node in the tree, the values of its entries are the largest values of the corresponding child nodes of P . Furthermore, in a quad-B tree, it is always true that the values in the nodes at the same level follow an ascending order from left-to-right. Let S decide to visit node P_n (a child node of P) whose corresponding entry in P is (P_n, V_n) .

If P_n has not been modified, S will arrive at P_n correctly. Suppose P_n has been split because of a previous insertion. When it happens, the new largest value in P_n will be V'_n ($\leq V_n$). If $V'_n = V_n$, then S arrives at P_n as intended. If $V'_n < V_n$, then S can use the link pointer to find the correct node whose largest value is greater than or equal to V_n . This is possible because a link pointer is introduced simultaneously at the time of splitting a node.

We also need to consider the case where P_n is deleted. With the mark-and-remove approach, we put a pointer in P_n to reference the head link pointer at the level of P_n . As

values in the nodes at the same level follow an ascending order from left-to-right, S can traverse via the link pointers until there is a node whose largest value is greater than or equal to V_n . Therefore, a search operation can always continue its downward traversal at each level of the quad-B tree. \square

Theorem 4.9 *The protocol QBL is deadlock and livelock free.*

Proof: Assume that a set of operations is deadlocked under the protocol QBL, and let Δ be the set of all deadlocked operations. We derive a contradiction out of this assumption. Let T be the oldest operation in Δ .

- **a:** T is a search operation. Since T only acquires a single lock at each step, it cannot be blocked forever. Furthermore, livelock of a search operation is not possible, because update operations are adopting the lock-coupling approach. The horizontal chase of link pointers never goes beyond one extra node as only one update operation can be active within its scope.
- **b:** T is an insert operation in its first phase. Since T used ϵ -lock-coupling, no operation can overtake T . Thus, T cannot be blocked forever, a contradiction.
- **c:** T is an insert operation in its second phase, executing *QbL.CleanUp*. In this case, T is trying to upgrade an ω -lock to an ϵ -lock. T cannot be in conflict with any other update operation to upgrade the lock because no update operation can be within its scope. A younger search operation may have placed a χ -lock on the node which T is acquiring, but it will eventually terminate because of the order of lock acquisition/upgrade (top-down and left-to-right fashion). Note that individual subsearch operations belonging to a search operation can proceed independently, since they are executed in parallel. Thus, T cannot be blocked forever by search operations. Hence, a contradiction to the assumption.
- **d:** T is a delete operation in its first phase. The argument is the same as in (b).
- **e:** T is a delete operation in its second phase. The argument is the same as in (c). \square

Theorem 4.10 *The protocol QBL based on the link technique is correct.*

Proof: From Theorem 4.9, QBL is deadlock free.

From the lemma above, there are two overtaking situations. The first situation is where an insert operation (I) overtakes a search operation (S). In our implementation, at the leaf nodes of a quad-B tree, each object has a timestamp. It is the timestamp of an insert operation which inserted the object. There is also a timestamp for each window search operation. By using the timestamps, objects which are added by younger operations are hidden from an older search operation. Hence, the overtaking does not affect the results of S . Therefore, from the perspective of S , the overtaking does not take place. The second situation is where a delete operation (D) overtakes a search operation (S). This only happens when S has a single search path (i.e., there is no sub-search). Hence, D and S can be ordered serially according to the times they access a leaf node and the overtaking does not cause any incorrect results for S .

Using the Lemma 4.11, a search operation always find a node to continue its downward traversal. Therefore, it will arrive at the leaf nodes and see the available objects. The only exception is when there are insert operations overtaking the search operation. As discussed in the previous paragraph, a search operation will not see the objects which are added by younger insert operations. Hence, a search operation will not see any object which is not available to it.

From the above discussion, we conclude that the protocol is correct. \square

Chapter 5

Rectangular Data

5.1 R-Simple Tree

The **R-tree** proposed by Guttman [Gut84] is the first index structure we study in this chapter. We will extend the R-tree with five different approaches to support concurrent operations accessing the tree. In this section, we discuss the **R-simple** tree which is a modified R-tree to be used with the **simple-lock** locking method.

An R-tree is a hierarchical data structure derived from the B-tree. Each non-leaf node in an R-tree represents the smallest d -dimensional rectangle that encloses the rectangles represented by its child nodes. The leaf nodes contain pointers to the actual geometric objects in the database. In an R-tree, rectangles associated with different nodes may overlap. An object may be spatially contained in the rectangles associated with several nodes, yet it can only be represented by one leaf node. This means that processing a query often requires traversing several search paths before ascertaining the presence or absence of a particular object.

More formally, it is a height-balanced tree with all the leaf nodes at the same level. Each node in the tree has at least m entries and at most M entries ($1 \leq m < M$) where $m = \lceil M/2 \rceil$. The root may have fewer than m entries, but has at least two entries if it is a non-leaf node. Each non-leaf node of the tree contains entries of the form (MBR, P) , where MBR is the **minimum bounding rectangle**¹ associated with a child node P . A leaf node contains entries of the form (MBR, O) where MBR is the MBR associated with object O .

¹A minimum bounding rectangle of a node is the smallest rectangle which contains all the rectangles pointed to by the node.

With no modification at all to an R-tree data structure, we can allow concurrent access to it by associating a single lock with its root. In the R-simple tree, only the root of the tree has a lock associated with it. (This is equivalent to associating a lock with the entire tree.) The lock management is very simple. The only difference from sequential access is that multiple search operations are allowed using a share-lock (i.e., a search operation requires a ρ lock on the root). But for an insert or delete operation, an exclusive lock (the ϵ -lock) must be acquired, which block any conflicting access to the tree.

Our concurrency control protocol, RS, for R-simple trees is implemented by a collection of procedures, for search, insert, and delete operations, as well as those for tree-reorganization. We now show how these operations can be carried out on an R-simple tree.

5.1.1 Search

$Search(W, P)$ starts at the root of the tree and descends to the leaf level. When a search operation is in progress in an R-tree, updates to the tree are prevented by a ρ lock.

$Rs.Search(W, P)$

1. Acquire $\rho(P)$.
2. $Rs.DoSearch(W, P)$.
3. Release $\rho(P)$.

$Rs.DoSearch(W, P)$

- **if P is a non-leaf node then**
 - Let $\{E_i = (MBR(P_i), P_i) \mid i = 1, \dots, k\}$ be the entries in P such that $MBR(P_i) \cap W \neq \phi$.
 - **if $(k > 0)$ then**
 - * For each entry E_i found above, continue the search with $Rs.DoSearch(W \cap MBR(P_i), P_i)$.
- **if P is a leaf node then**
 - For all entries in P whose MBR's overlap with W , return their object ids. □

5.1.2 Insert

$Insert(O, P)$ need first acquire an ϵ lock on the root. Once it holds an ϵ lock, no other operation can access the tree.

$Rs.Insert(O, P)$

1. Let E_O be the entry representing the object O .
2. Acquire $\epsilon(P)$.
3. $L = Rs.Select(MBR(O), P)$, which selects a leaf node where object O is to be placed.
4. Add entry E_O to L .
5. $Rs.Update(L)$.
6. Release $\epsilon(P)$. □

$Rs.Select(W, P)$

1. **if** P is a non-leaf node **then**
 - Find an entry $E_s = (MBR(P_s), P_s)$ in P , where $MBR(P_s)$ requires the smallest enlargement to include W . Resolve ties by choosing E_s with the smallest $MBR(P_s)$.
 - Continue the selection with $Rs.Select(W, P_s)$.
2. **if** P is a leaf node **then** returns P . □

$Rs.Update(L)$

1. **if** $\neg FULL(L)$ **then** return.
2. Split L into L and L' , update $MBR(L)$, and add entry $(MBR(L'), L')$ to $PARENT(L)$.
(If L was the root, create a new root first and make it $PARENT(L)$.)
3. $Rs.Update(PARENT(L))$. □

5.1.3 Delete

$Delete(O, P)$ can be carried out in a manner similar to the insert operation. It first acquires an ϵ lock on the root to block all other operations, and then carries out its steps in two phases. The first phase is to locate the object with the procedure $FindLeaf(O)$. This procedure is similar to a search operation, except that it returns the leaf node which contains the object, O , to be deleted. The second phase removes the object, and reorganizes the tree if an underflow occurs.

5.1.4 Correctness of the Operations

The proof for the R-simple is straight forward as there can only be one update (either insert or delete) operation active at one time.

Lemma 5.1 *When an update operation is working in the R-simple tree, no other operation can be active.*

Proof: Because ϵ -locks are incompatible, after an update operation has ϵ -locked the root, other update operations will have to wait. Similarly, because ϵ -locks are incompatible with ρ -locks, a search operation will need to wait. \square

Theorem 5.1 *The R-simple tree is both deadlock and livelock free.*

Proof: Since there is only a single lock, there cannot be any deadlock nor livelock. \square

Theorem 5.2 *The protocol RS is correct.*

Proof: From Theorem 5.1, the R-simple tree is deadlock free. From Lemma 5.1, as an update operation cannot co-exist with any other operation. It can only be active before or after a search operation. Therefore, overtaking between update and search operations is not possible. Furthermore, because of the same reason, a search operation will always see its available objects. Hence, we conclude that the protocol RS is correct. \square

5.2 R-Lock Tree

Locking the entire tree whenever an updating operation is invoked reduces concurrency. Note that, when there is no splitting or merging of nodes in the tree, the tree remains

in a consistent state if individual nodes are locked while being accessed by different types of operations. The granularity of locking now becomes at the node level rather than the entire tree. This is the second locking method, called the **modify-lock** method, and the **R-lock** tree is a modified R-tree to be used with this method. It allows multiple concurrent operations accessing the tree. Either an insert or a delete operation can proceed together with search operations. If an update (either insert or delete) operation causes any overflow or underflow at nodes, we lock the whole tree until a splitting and/or merging process has been completed. To manage overflows, we add a buffer space in each leaf node to accommodate overflowed entries. We also make use of the *maintenance queue*, denoted Q , which records the nodes at which an overflow or underflow has occurred, until the tree is re-organized.

We will need to keep track of and control the number of concurrent search, insert, or delete operations operating on the tree. For this purpose, we introduce the *in-transaction counter*, denoted TC_{in} , which is incremented whenever a transaction invokes an operation on the R-tree. We also introduce the *out-transaction counter*, denoted TC_{out} , which is incremented whenever an operation on the tree is completed. Therefore, the difference, $TC_{in} - TC_{out}$, indicates the number of operations currently accessing the tree. We check for $TC_{in} - TC_{out} = 0$ to see if the whole tree can be locked for reorganization explained below.

A system process, *Maintain*, is used to reorganize the R-tree. It checks for entries in the maintenance queue Q and performs the splitting and merging of nodes. The *Maintain* process gets activated after overflows have occurred as a result of insert operations, or underflows have occurred as a result of delete operations.

We lock individual nodes using a ρ lock or ϵ lock. Each operation must request locks on nodes in a top-down and left-to-right fashion; i.e., after locking a node, it cannot request a lock on a node that is closer to the root in the above order. This order of lock requests guarantees deadlock freedom.

In the R-lock tree, overtaking of operations is allowed. Hence, when there are more than one update operations active in the tree, the serialization problem described in Chapter 4 may occur. To solve the problem, the R-lock allows only one update operation (either insertion or deletion) active in the tree. A lockable variable, U_l , is used for the purpose. Every update operation will need to acquire an ϵ -lock on U_l before proceeding its work. The lock is released after the operation has made the changes at a leaf node.

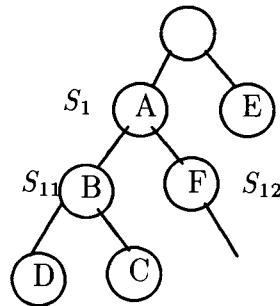


Figure 5.1: Search operations using the modify-lock method.

Our concurrency control protocol, RM, for the R-lock trees is implemented by a collection of procedures, for search, insert, and delete operations, as well as those for tree-reorganization. In the following subsections, we discuss them in detail.

5.2.1 Search

As in the simple-lock method, a search operation starts at the root of the tree and descends to the leaf level. In order to access a node, a ρ lock on the node must be obtained to avoid any modification to it by an update operation (i.e., insert or delete). In Figure 5.1, the search operation S_1 holds a ρ lock on node A . It reads the content of A and starts two sub-search operations, S_{11} and S_{12} . The ρ -lock on A can now be released because there cannot be any split or merge in nodes A , B or F , while S_{11} and S_{12} are in progress. At most one ρ -lock is needed at each branch of the search operation.

More formally, $Search(W, P)$ under the modify-lock method is carried out as follows.

Rm.Search(W, P)

1. Acquire $\epsilon(TC_{in})$, increment TC_{in} by 1, then release $\epsilon(TC_{in})$.
2. *Rm.Look*(W, P).
3. Acquire $\epsilon(TC_{out})$, increment TC_{out} by 1, then release $\epsilon(TC_{out})$. □

Rm.Look(W, P)

- Acquire $\rho(P)$.

- **if** P is a non-leaf node **then**

- Let $\{E_i = (MBR(P_i), P_i) \mid i = 1, \dots, k\}$ be the entries in P such that $MBR(P_i) \cap W \neq \phi$.
- Release $\rho(P)$.
- For all entries E_i found above, continue the search in parallel invoking $Rm.Look(W \cap MBR(P_i), P_i)$.

- **if** P is a leaf node **then**

- For all objects whose MBR's overlap with W , return their object ids.
- Release $\rho(P)$. □

5.2.2 Insert

Insertion under the modify-lock method can be carried out much as under the simple-lock method, except for the locking operations involved. At the beginning, an insert operation first acquires an exclusive lock on U_l to avoid any other update operation working in the tree. In descending down the tree, at each visited node, it acquires an ϵ lock, update the MBR of the node if needed, then choose the next node to visit, and finally release the lock. The reason why we update the MBR of a visited node in the first phase is to allow subsequent search operations to find the newly inserted object.

After adding the new object at a leaf node, the exclusive lock on U_l is then released to allow other update operations to proceed. If no overflow occurs at the leaf node, the second phase is not necessary. As we mentioned earlier, we permit temporary overflows and delay the second phase of an insert operation by simply adding a new entry to the maintenance queue, Q . The *Maintain* process will use Q to reorganize the tree periodically. More formally, $Insert(O, P)$ under the modify-lock method is carried out as follows.

$Rm.Insert(O, P)$

1. Let E_O be the entry representing object O .
2. Acquire $\epsilon(U_l)$.
3. Acquire $\epsilon(TC_{in})$, increment TC_{in} by 1, then release $\epsilon(TC_{in})$.
4. $L := Rm.Select(MBR(O), P)$ to search for a leaf node to which to add the object O .

5. Acquire $\epsilon(L)$, add entry E_O to L , then release $\epsilon(L)$.
6. Release $\epsilon(U_l)$.
7. **if** $OVERFL(L)$ **then**
 - Acquire $\epsilon(Q)$, $APPEND(Q, L)$, then release $\epsilon(Q)$.
8. Acquire $\epsilon(TC_{out})$, increment TC_{out} by 1, then release $\epsilon(TC_{out})$. □

Rm.Select(W, P)

1. Acquire $\rho(P)$.
2. **if** $MBR(P)$ does not contain W **then**
 - Upgrade $\rho(P)$ to $\epsilon(P)$
 - Enlarge $MBR(P)$ to accommodate W .
 - Downgrade $\epsilon(P)$ to $\rho(P)$.
3. **if** P is a leaf node **then** release $\rho(P)$ and return P .
else
 - Find an entry $E_s = (MBR(P_s), P_s)$ in P , where $MBR(P_s)$ requires the smallest enlargement to include W . Resolve ties by choosing E_s with the smallest $MBR(P_s)$.
 - Release $\rho(P)$.
 - Continue the selection with $Rm.Select(W, P_s)$. □

5.2.3 Delete

We carry out the delete operation similarly to the R-simple tree. At the beginning, a delete operation first acquires an exclusive lock on U_l to avoid any other update operation working in the tree. We use the same queue, Q , which is used for insert operations, to record the nodes which have been deleted or the nodes whose MBRs should be shrunk. If the size of Q exceeds a certain limit, then the *Maintain* process recomputes the affected MBR's.

After removing the object from a leaf node, the exclusive lock on U_l is released to allow other update operations to proceed. A reorganization of the R-lock tree may be

necessitated by underflows. When an underflow occurs at some node, we append the node to the maintenance queue, Q . As we discuss in the next subsection, this will be detected by the *Maintain* process, which is constantly checking if Q is empty.

More formally, $Delete(O, P)$ under the modify-lock method is carried out as follows.

Rm.Delete(O, P)

1. Acquire $\epsilon(U_i)$.
2. Acquire $\epsilon(TC_{in})$, increment TC_{in} by 1, then release $\epsilon(TC_{in})$.
3. $L := Rm.Find(MBR(O), P)^2$ to search for the leaf node containing the object O .
4. $MBR.old := MBR(L)$.
5. Acquire $\epsilon(L)$, remove the entry representing object O from L , and set $MBR(L)$ to the MBR of the resulting L .
6. Release $\epsilon(U_i)$.
7. **if** ($UNDERFL(L)$) or ($MBR.old \neq MBR(L)$) **then**
 - Acquire $\epsilon(Q)$, $APPEND(Q, L)$, then release $\epsilon(Q)$.
8. Acquire $\epsilon(TC_{out})$, increment TC_{out} by 1, then release $\epsilon(TC_{out})$. □

5.2.4 Maintain

The system process *Maintain* checks the maintenance queue Q periodically. When Q becomes non-empty, *Maintain* prepares for reorganization of the R-tree by waiting until no operation is accessing the tree. For this purpose, it first acquires and holds an exclusive lock on TC_{in} . This prevents other transactions from starting a new access to the R-tree. It then compares the values of TC_{in} and TC_{out} . If they are the same, then all previous operations accessing the tree have completed and it can proceed to do the reorganization. Otherwise, it releases the lock on TC_{out} and tries again later.

When all previous operations have completed, *Maintain* then takes nodes from Q one after another and processes them accordingly to the reason why they are in the queue (overflow, underflow, shrunken MBR). When all the nodes in Q have been processed, *Maintain* releases the locks on TC_{in} and TC_{out} to allow resumption of operations on the R-tree.

²The operations of *Rm.Find* is the same as *Rm.Look* except it returns the node which contains object O .

Rm.Maintain()

• **while** (*TRUE*) **do**

1. Acquire $\rho(Q)$.

2. **while** ($Size(Q) < Q_{max}$) **do**

{Release $\rho(Q)$, Sleep a while, Acquire $\rho(Q)$ } **End do**

3. Release $\rho(Q)$.

4. Acquire $\epsilon(TC_{in})$.

5. Acquire $\epsilon(TC_{out})$.

6. **if** $TC_{in} = TC_{out}$ **then**

– Reset $TC_{in} := TC_{out} := 0$.

– $P :=$ The first node in Q .

– **while** ($P \neq NULL$) **do**

* **if** *UNDERFL*(P) **then** re-distribute entries of P into its siblings.

* **if** *PARENT*(P) underflows as a result **then** *APPEND*($Q, PARENT(P)$).

* **if** *OVERFL*(P) **then** split P into P and P' . Add a new entry for P' to *PARENT*(P).

* **if** *PARENT*(P) overflows **then** *APPEND*($Q, PARENT(P)$).

* $MBR.old := MBR(PARENT(P))$.

* Update the MBR's of P , its siblings and *PARENT*(P), if necessary.

* **if** $MBR(PARENT(P)) \neq MBR.old$ **then** *APPEND*($Q, PARENT(P)$).

* $P := NEXT(Q)$.

end do.

else

– Release $\epsilon(TC_{out})$

– Sleep a while.

– Go back to step 5.

7. Release $\epsilon(TC_{in})$ and $\epsilon(TC_{out})$.

end do.

□

5.2.5 Correctness of the Operations

In this section, we will show that the protocol RM is correct. In the proof of the correctness theorem (Theorem 5.4) given below, we use the concept of serializability [BHG87] instead of the non-overtaking condition stated in Section 4.1.5.

Lemma 5.2 *There can be multiple search operations but a single update operation active in the tree at the same time.*

Proof: The in-transaction counter is available to all operations when the tree is not reorganizing. Multiple search operations can be active in the tree after they have incremented the counter. As each operation, except the maintain operation, will only use the in-transaction counter for a short time, it will not block any other operation to become active in the tree.

On the other hand, an update operation acquires an $\epsilon(U_i)$ before it starts working in the tree. The lock will not be released until the update operation finishes. Therefore, there cannot be a second update operation active in the tree. \square

Lemma 5.3 *There is no deadlock between a maintain operation with any other operation.*

Proof: The maintain operation is invoked periodically to check the size of the maintain queue, Q . It holds a lock on Q temporarily to find out its size and quickly releases the lock afterwards.

When the size of Q is large enough, the maintain operation acquires locks on the transaction counters. It waits until there is no operation active in the tree. It acquires and releases the out-transaction counter repeatedly until the out-transaction counter has the same value as the in-transaction counter. During that time, the maintain operation does not lock any node in the tree. Hence, deadlock between the maintain operation and other operations is not possible. If all active operations run at a fair rate and will not be aborted, they will be able to finish and update the out-transaction counter afterwards. Once the counters have the same value, the maintain operation is the only operation working on the tree and deadlock is not possible. \square

Theorem 5.3 *The R-lock tree is both deadlock and livelock free.*

Proof: Each operation, either a search or an update, acquires one lock at a time. Therefore, they cannot be blocking each other. From Lemma 5.3, active search and update operations

cannot be blocked by the maintain operation. During the reorganization of the tree, there can only be one maintain operation active. Therefore, the R-lock tree is deadlock and livelock free. \square

Theorem 5.4 *The protocol RM based on the lock-modify protocol is correct.*

Proof: When there are only search operations working in the tree, a search operation will always see the available objects. When there is an insert operation active, it expands the MBRs of the nodes along its access path as needed. On the other hand, a delete operation does not change the MBRs of the nodes at all. Hence, there is no shrinkage of MBRs and any later search operations will not miss any available object.

When an update operation and a search operation accessing the same leaf node, their execution order is dictated by the times they acquire a lock on the node. Furthermore, as there is only one active update operation working on the leaf nodes at a time, the serializable schedule of the operations is the arrival times of operations reaching the common leaf nodes.

\square

5.3 R-couple Tree

The **R-couple** tree is a modified R-tree to be used with the lock-coupling approach. In the tree, we lock individual nodes rather than the whole tree. With this locking method, a search or an update operation locks children of a node before releasing its lock on the node itself. Thus, a process always holds at least one lock while accessing an R-tree, but locks are acquired locally in different sub-trees of the R-tree. Multiple search, insert, and delete operations can be taking place concurrently in different parts of the same tree, regardless of overflows and underflows at nodes.

Our concurrency control protocol, RC, for R-couple trees is implemented by a collection of procedures, for search, insert, and delete operations, as well as those for tree-reorganization. The implementation of the procedures is the same as that in the R-tree in Chapter 4. Hence, the details of the procedures are skipped here.

5.3.1 Search

A search operation starts at the root of a tree and descends to the leaf level. At each visited node, a ρ lock on it must be acquired and held until the ρ locks on the child nodes to be

visited are acquired. If the ρ lock on the parent is released after a ρ lock on the first child has acquired, the parent may be updated before a ρ lock on the second child is acquired. This will cause an incorrect result. To implement this rule, we introduce a lockable variable, *Count*, which is initialized to the number of subsearch operations initiated at a node. It is decreased by one whenever a child node is ρ -locked. When it reaches 0, the ρ lock on the parent is released.

More formally, $Search(W, P)$ under the lock-coupling approach is carried out by calling $Rc.Search(W, P, Count)$.³

5.3.2 Insert

As with the insert operation in the simple-lock approach, we carry out an insert operation in two phases. During the first phase, we locate the leaf node to which to add the new object. The path in the tree we take is called the **insertion path**. After adding the entry representing a new object to a leaf node (i.e., in the second phase), we may have to perform some maintenance operations by tracing back the insertion path to

1. reorganization of the tree, and/or
2. enlarge the MBR's of some nodes.

In the modify-lock approach discussed in the previous section, we delegated these maintenance chores to a system process, called *Maintain*, which periodically carried them out. This meant that they were carried out after some delays. In this section, we take a different approach and carry them out as a part of the insert operation without help from a system process.

More formally, an insert operation is carried out by $Rc.Insert(O, P)$ where O is the object to be inserted and P is the root of the tree initially.⁴

5.3.3 Delete

We use two phases to carry out a delete operation. The first phase is to locate the leaf node which points to the object to be deleted. This phase is almost identical to $Rc.Search$. (Note that the condition $W \subseteq MBR(P_i)$ should be used here to find the next visiting nodes,

³ $Rc.Search$ is implemented the same as $Rt.Search$ in Section 4.2.1.

⁴ $Rc.Insert$ is implemented the same as $Rt.Insert$ in Section 4.2.2.

instead of $MBR(P_i) \cap W \neq \phi$.) Only ρ locks are held on at most two levels of non-leaf nodes, and MBR's are not shrunk in this phase. Once the object has been found, it is removed and the tree is reorganized in the second phase.

In the second phase, we backtrack the R-tree if needed. In the R-couple tree, we face the same two problems as in the R-tree in Chapter 4. The first problem is how to deal with concurrent insertions and changing parent pointers. The second problem is the potentially conflicting lock requests with other updating operations. We use the same solution in Section 4.3.3 and apply them to the R-couple tree.

The implementation of $Rc.Delete(O, P)$ is the same as $Rt.Delete$ in Chapter 4.

5.3.4 Correctness of the Operations

The proofs of correctness of the operations are the same as those in the R-tree in Chapter 4. Therefore, we will only state a lemma and two theorems without proof.

Lemma 5.4 *A search operation cannot overtake an update operation if they are traversing the same path. Similarly, an update operation cannot overtake any other operation.*

Theorem 5.5 *The R-couple tree is both deadlock and livelock free.*

Theorem 5.6 *The protocol RC is correct.*

5.4 R-opt Tree

In the previous sections, we have discussed three different approaches to support concurrent operations in an R-tree. The three approaches try to avoid conflicting situations between different operations and are pessimistic in nature. In this section, we will describe an optimistic approach to support concurrent operations. In the new approach, search operations check its correctness before committing their operations. The validations are done both at leaf nodes and internal nodes. As for an internal node, the validation is done with the range information. At a leaf node, the validation is done with the range information and the timestamp of the node. When the validation fails, a search operation would need to re-start.

Shasha [ShG88] suggested the *give-up* technique with the help of redundancy introduced by a **range** field. Before any operation can work on a node, the protocol checks the range of

the node. If the operation's argument is not in the range, he gives up and tries an ancestor of that node. In the case of a B-tree, the range information is an interval of scalar values. However, an R-tree deals with 2-dimensional objects and there is no specific ordering among the nodes. The range information of a node, P , can be considered as a flag indicating if a change has happened at P . There are seven possible changes to P :

- 1: An expansion of its MBR.
- 2: A shrinkage of its MBR.
- 3: Addition of a new entry.
- 4: Removal of an entry.
- 5: Splitting.
- 6: Merging.
- 7: Marking it as deleted.

When we visit a node P while carrying out an operation, we would need to determine if backtracking is necessary because of a recent change of P . In Figure 5.2, we list out changes which will affect subsequent actions of an operation. Insert and delete operations use the log table during their first phase of work. To minimize the storage requirement of bookkeeping, we add only two extra pieces of data, the **version number** and the **change type** for each node in an R-tree. When a node is first created, its version number is 0 and the change type is *NULL*. Whenever a change is made to the node, the version number is incremented by one and the type of modification is recorded in the change type. When the change is reflected in the parent node, the parent copies the new version number of the updated child. During a visit to a node, an operation compares the version number provided by the parent to the current version number of the node. If they are different, the change type field is used to determine if backtracking is necessary. The data structure of each entry in an R-tree node is shown in Figure 5.3.

In the give-up approach, overtaking amongst different operations are allowed. An operation releases a lock on its current node before it visits the next one. We use the timestamp technique as in the quad-B tree to solve the serialization problem at the leaf nodes. At each leaf node, we add one additional data which stores the timestamp when the node was last

	Search	Insert	Delete
MBR expanded	-	-	-
MBR shrank	-	-	-
Added entry	-	-	-
Removed entry	-	-	-
Split	*	*	*
Merge	*	*	*
Mark	*	*	*

Figure 5.2: Node changes and their effects to operations. (* - Changes which will affect subsequent operations).

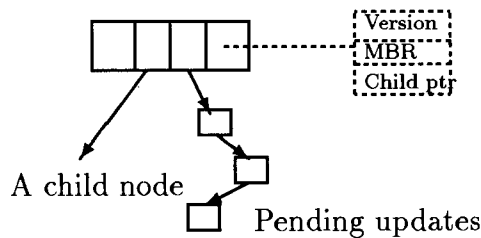


Figure 5.3: A node in the R-opt tree.

modified. This timestamp only affects search operations. When a search operation detects an inconsistency at a leaf node, it backtracks and re-starts its operation.

Our concurrency control protocol, RG, for R-opt trees is implemented by a collection of procedures, for search, insert, and delete operations, as well as those for tree-reorganization. In the following subsections, we discuss them in detail.

5.4.1 Search

As always, a search operation starts at the root of a tree and descends down to the leaf level. At each visited node, a ρ lock on it must be acquired. If there is a change in the node and the search is no longer in *range*, it will backtrack to an ancestor node. Consider the example in Figure 5.4, where a search operation S branches into three sub-search operations, S_1 , S_2 , and S_3 at node D . If at node P_2 , S_2 noticed a change, it would go backward to an ancestor node. It may go back as far as node D . At the node, the new S_2 may branch off three

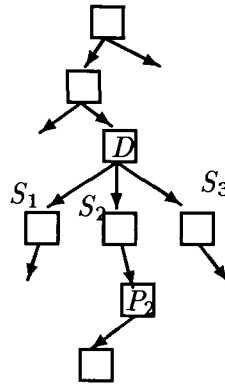


Figure 5.4: Search operation in an R-opt tree.

sub-searches even if S_1 and S_3 complete. If there are updates in between the completion of S_1 , S_3 and the three new sub-searches, there will be inconsistency in the results. If we only want to restrict the new S_2 to traverse its previous path only, this requires us to record all the relevant logs and the information can be huge and is similar to a tree of logs. During the development of the R-opt tree, we argue that update operations are infrequent, therefore a simple log table would be desired. Before we describe our solution, we give the following definitions.

Branch Node(BN): A node is a *branch node* of a search operation S if it is the first node visited by S where there are multiple entries in the node whose MBRs overlap with the search window. I.e., more than one sub-search starts after visiting the node for S .

Stable Node(SN): A node is *stable* to a search operation S if the version number of the node is the same during different visits by S .

To maintain the *range* information, a log table as illustrated in Figure 5.5 is used for each search operation. Each row in the log table is a tuple of two values as (node_id, version_number). It records the *version numbers* of all the nodes that are visited by a search operation. During the downward traversal of a search operation, if it notices the version number of the current node is different from what it obtains from the node's parent, the current node might have been changed. Depending on the change type which is stored

Node Id	Version Number
A	0
B	1
C	1
D	2
....

Figure 5.5: Log Table.

in the node, the search operation starts backtracking at its branch node until a stable node is located. Let the stable node be G . At node G , the search operation aborts all the sub-searches and purges its results. It then resumes and continues the search downward. In Figure 5.4, S_2 backtracks to its branch node D which incidentally is also its first stable node; otherwise, it continues backtracking until a stable node is reached.

The log table of a search operation only provides the information to determine if an ex-visited node has been changed or not. In order to backtrack, each node will need to know how to access its ancestors. We can use a stack for each search path, but several or many stacks will be needed because of multiple sub-searches. In our algorithms, we include a parent pointer in each node to provide a route for backtracking. However, as a search operation only locks a node at a time, it may have a new parent before the operation accesses its parent. Therefore, the procedure *GetParent* repeatedly uses the parent pointer of a node until there is no discrepancy.

If an internal node visited by a search operation is newly modified, the search operation backtracks only when the modification will affect its later searches. However, if the node is a leaf node, not only the modification will affect the search result, but there is a possibility of having the serialization problem as well. When a search operation has no branch node, the problem cannot occur. Therefore the timestamps are not used. The timestamps at the leaf nodes are needed when a search operation generates multiple sub-searches. If the search operation visits a leaf node whose timestamp is younger than that of the operation, then there must be an update operation overtaking the search operation. There is also a queue, DQ , which contains the timestamps of all un-finished update operations. At a leaf node, if the search operation's timestamp is greater than any timestamp in DQ , then the search operation has overtaken an update operation. In both cases, the search operation backtracks

to a stable node. When it traverses down again, its timestamp is reset. More formally, Search under the give-up approach is carried out by calling $Rg.Search(W, P, lastset, Bnode, Stime)$, where $lastset$ is the version number of P from P 's parent and $Bnode$ is the branch node of the operation, and $Stime$ is the starting time of the search operation. Initially, $lastset$ and $Bnode$ are both $NULL$. Each search operation will have its own log table (LOG) which is shared by all its sub-search operations. For the algorithm presented below, $TIMPSTAMP(P)$ returns the timestamp of node P and $FTIME(DQ)$ returns the time of the oldest element in queue DQ .

$Rg.Search(W, P, lastset, Bnode, Stime)$

1. Acquire $\rho(P)$.
2. **if** ($lastset = NULL$) or ($ctype(P, lastset, SEARCH)^5 = FALSE$) **then**
 - Acquire $\rho(Bnode)$.
 - **if** P is a non-leaf node **then**
 - Let $\{E_i = (MBR(P_i), P_i) \mid i = 1, \dots, k\}$ be the entries in P such that $MBR(P_i) \cap W \neq \phi$.
 - **if** ($k > 0$) **then**
 - * **if** ($Bnode = NULL$) and ($k > 1$) **then**
 - Acquire $\epsilon(Bnode)$.
 - $Bnode := P$.
 - Downgrade $\epsilon(Bnode)$ to $\rho(Bnode)$.
 - else if** ($Bnode = NULL$) **then**
 - **if** P is in the log table **then** update the version number of P in the log table.
 - else** insert $(P, version(P))$ into the log table.
 - * For all entries E_i found above, continue the search in parallel invoking $Rg.Search(W \cap MBR(P_i), P_i, lastset_i, Bnode, Stime)$ where $lastset_i$
 - Release $\rho(Bnode)$.
 - Release $\rho(P)$.
 - Return.

⁵A predicate to tell if the previous modification in P affects a given operation.

else if ($(Bnode = NULL)$ or
 $((TIMESTAMP(P) > Stime)$ and $(FTIME(DQ) < Stime)$) **then**

- For all objects whose bounding rectangles overlap with W return their object ids.
- Release $\rho(P)$.
- Return.

else release $\rho(P)$.

3. Release $\rho(P)$.

4. $F := Bnode$.

5. Acquire $\rho(Bnode)$.

6. **while** ($F \neq NULL$) and (version(F) \neq LOG(F).version⁶) **do**

- $F := GetParent(F)$.

enddo

7. **if** ($F = NULL$) **then** $F := Root$.⁷

8. Abort all sub-searches underneath F and purge their results.

9. $Bnode := NULL$.

10. $Stime := Current\ system\ time$.

11. $Rg.Search(W, F, NULL, Bnode, Stime)$. □

When the procedure *GetParent* is called, node L is ρ -locked. The procedure will read the parent pointer of L , releases the lock on it and attempts to acquire a lock on the parent of L . If the new node is indeed the parent of L , the procedure returns the reference and maintains the ρ -lock on the parent. Otherwise, it repeatedly reads the pointer of L until the correct parent is found. Even when L is deleted, because of the mark-and-remove approach for deletion, the parent will be found.

GetParent(L)

1. **if** ($L = NULL$) **then** return $NULL$.

⁶The recorded version number of node F in the log table.

⁷The root of the tree.

2. $L_p := PARENT(L)$.
3. Release $\rho(L)$.
4. **if** ($L_p = NULL$) **then** return $NULL$.
5. $\rho(L_p)$.
6. **while** ($L_p \neq NULL$) and (L is not a child of L_p) **do**
 - Release $\rho(L_p)$.
 - Sleep a while.
 - $\rho(L)$.
 - $L_p := PARENT(L)$.
 - **if** $DELETE(L)$ **then**
 - Acquire $\rho(L_p)$.
 - Return L_p .
 - Release $\rho(L)$.
 - **if** ($L_p = NULL$) **then** return $NULL$.
 - $\rho(L_p)$.
- enddo**
7. Return L_p . □

5.4.2 Insert

The insert operation works similarly to that in a B^+ -tree. After inserting an object in a leaf node, if the node overflows, it would then be split, and the split propagates up the tree. The first phase is to select a leaf node to add the new object. The second phase is to insert the object at the leaf level, and propagate the required updates upward. However, at the time when the object is added, there can be changes in higher level which affect the parental relationships among nodes.

In the first phase, the MBR's of some nodes will change if the object to be inserted covers additional area. If the bounding rectangle of a node is updated immediately, it may be changed by a subsequent delete operation (see the discussion on delete operation),

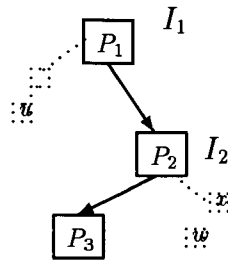


Figure 5.6: Phase one of inserting an object.

which may cause a later search operation to have an incorrect answer. This problem is solved by delaying the update; instead of updating the MBR of the visited node immediately, we record the MBR of the object to be inserted in a data structure, called *pending update* (see Figure 5.3), associated with the node. We call the MBR stored in pending update a **pending MBR**. The pending MBR is read during the second phase to enlarge the node's bounding rectangle. In the mean time, before the second phase of the insertion is completed, subsequent search operations look at both the bounding rectangle and the pending MBR of the node. (The search procedure described in the previous section should be modified accordingly.) Incidentally, observe that, if a node has a pending MBR, then all its descendants along the insertion path will also have the same pending MBR.

In some occasions an insert operation has been adding pending updates at high levels. However, on its way towards the leaf level, it finds out the lower level nodes have larger bounding rectangles because of previous insert operations. In order to ensure the updating phase will remove all the pending updates, the insert operation continues to add pending updates once it has started. The steps can be illustrated by using Figure 5.6. In the figure, an insert operation I_1 visits P_1 , it first acquires a ρ -lock on P_1 . As the new object (O) is outside the *MBR* of P_1 , it upgrades the lock to an ϵ -lock and appends a new pending update U . The operation then releases the lock and visits P_2 . At P_2 , another insert operation (say I_2) is working in its second phase and it has just removed its pending update w . This would make the *MBR* of P_2 larger than what I_1 has seen. When I_1 visits P_2 , O is inside *MBR* of P_2 . However, because of the previous pending update u , I_1 will continue adding a pending update, x , at P_2 .

If there is no overflow at the leaf node, the insert operation will only backtrack, update

bounding rectangles and remove the pending updates. To combine the effort of tree reorganization amongst insertions and deletions, we permit temporary overflows. The *maintain* operation is used to handle the splitting and updating of nodes. When a node is to be split, its reference is added to the maintenance queue Q . The *Maintain* operation will use Q to reorganize the tree. During the reorganization, when there is a pending update associated with a node, the node will not be deleted. More formally, *Insert* has three parameters. The first one is O which is the object to be inserted, P is initially set to the root of the tree, and $SeqNo$ is a unique sequence number assigned to the operation.

$Rg.Insert(O, P, SeqNo)$

1. $Curtime := \text{current system time.}$
2. $SeqNo := \text{transaction number for the insert operation.}$
3. Acquire $\epsilon(DQ)$.
4. Add element $(SeqNo, Curtime)$ to DQ .
5. Release $\epsilon(DQ)$.
6. $L := Rg.Select(MBR(O), P, SeqNo)$, which returns a leaf node, L , in which to place object O . Let L_c be the entry referencing the object.
7. Acquire $\epsilon(L)$.
8. Insert L_c into L and update $MBR(L)$.
9. $TIMESTAMP(L) := \text{Max}(TIMESTAMP(L), Curtime)$.
10. Acquire $\epsilon(DQ)$.
11. Remove element $(SeqNo, Curtime)$ from DQ .
12. Release $\epsilon(DQ)$.
13. **if** $OVERFL(L)$ **then**
 - $\epsilon(Q)$.
 - $APPEND(Q, L)$.
 - Release $\epsilon(Q)$.
14. Release $\epsilon(L)$.
15. $Rg.Update(L, SeqNo)$. □

The procedure *Rg.Select* locates a leaf node to insert a new object. Its implementation is similar to the search operation. However, it only backtracks to its parent when a visited node is marked as deleted.

Rg.Select($W, P, SeqNo$)

1. **if** ($P = NULL$) **then** creates and returns P .
 2. Acquire $\rho(P)$.
 3. **if** *DELETE*(P) **then**
 - $L_p := PARENT(P)$.
 - *Rg.Select*($W, L_p, SeqNo$).
 - Return.
 4. **if** P is a non-leaf node **then**
 - Find an entry $E_s = (MBR(P_s), P_s)$ in P , where $MBR(P_s)$ requires the smallest enlargement to include W . Resolve ties by choosing E_s with the smallest $MBR(P_s)$.
 - **if** (an enlargement is required in E_s) or (a pending update has added at high levels) **then**
 - $\epsilon(P)$.
 - Append a pending update to node P . It contains the bounding information W and the transaction number.
 - Downgrade to $\rho(P)$.
 - Release $\rho(P)$.
 - *Rg.Select*(W, P_s).
- else** return P . □

The procedure *Rg.Update* is invoked to perform the second phase of an insert operation partially. When a leaf node is overflowed, its reference will be appended to the maintenance queue. At a later time, the system operation, *Maintain*, will use the queue to perform the necessary reorganization of the tree. The *Rg.Update* removes the pending updates of an insert operation deposited in its first phase. It adjusts the MBRs of nodes along the insert

path. The procedure *UGetParent* is invoked to obtain the parent of a node in order to backtrack.

Rg.Update(L, SeqNo)

1. Acquire $\rho(L)$.
2. $L_p := UGetParent(L)$.
3. **while** ($L_p \neq NULL$)
 - Acquire $\epsilon(L_p)$.
 - Let E_u be the entry in L_p referencing L .
 - **if** there is no pending update with the same *SeqNo* **then**
 - Release $\epsilon(L_p)$.
 - **Return**.
 - At L_p , adjust its MBR and that of the entry E_u .
 - Remove the pending update in L_p .
 - Downgrade $\epsilon(L_p)$ to $\rho(L_p)$.
 - $L_p := UGetParent(L_p)$.

enddo

□

The procedure *UGetParent* works similar as the procedure *GetParent* except it returns *NULL* when the current node L is found to be deleted.

UGetParent(L)

1. **if** ($L = NULL$) **then** return *NULL*.
2. $L_p := PARENT(L)$.
3. Release $\rho(L)$.
4. **if** ($L_p = NULL$) **then** return *NULL*.
5. $\rho(L_p)$.
6. **while** ($L_p \neq NULL$) and (L is not a child of L_p) **do**

- Release $\rho(L_p)$.
 - Sleep a while.
 - $\rho(L)$.
 - $L_p := \text{PARENT}(L)$.
 - **if** $\text{DELETE}(L)$ **then**
 - Release $\rho(L)$.
 - Return NULL .
 - Release $\rho(L)$.
 - **if** $(L_p = \text{NULL})$ **then** return NULL .
 - $\rho(L_p)$.
- enddo**

7. Return L_p . □

5.4.3 Delete

Similarly to the *insert* operation, there are two phases for the *delete* operation. In the first phase, it tries to locate the leaf node which contains the object. It uses the same procedure as in the *search* operation. However, once the leaf node is identified, all other sub-operations can be aborted.

After the removal of the object, the second phase is to update the MBR's and backtrack to the root if needed. The steps are similar to the steps in the second phase of an insert operation. It uses the *UGetParent* routine and quits at the node whose MBR does not shrink because of the deletion.

Besides updating the MBRs, a reorganization of the R-tree may be necessitated by underflows. When an underflow occurs at a leaf node after an object's removal, we append its reference to the maintenance queue, Q . The *maintain* operation will later reorganize the entries in the node. More formally, $\text{Delete}(O, P)$ is carried out as follows. O is the object to be deleted and P is the root of the tree.

Rg.Delete(O, P)

1. $\text{Curtime} := \text{current system time}$.

2. $SeqNo :=$ transaction number for the delete operation.
3. Acquire $\epsilon(DQ)$.
4. Add an new element $(SeqNo, Curtime)$ to DQ .
5. Release $\epsilon(DQ)$.
6. $L := Rg.Find(P, O, lastset, Bnode, Stime)$ to search for the leaf node containing the object O where $lastset$ and $Bnode$ are $NULL$ initially.
7. Acquire $\epsilon(L)$.
8. Remove object O , and the entry referencing O from L .
9. Update $MBR(L)$ and its version number and change type.
10. $TIMESTAMP(L) := Max(TIMESTAMP(L), Curtime)$.
11. Acquire $\epsilon(DQ)$.
12. Remove the element $(SeqNo, Curtime)$ from DQ .
13. Release $\epsilon(DQ)$.
14. Downgrade to $\rho(L)$.
15. **if** $UNDERFL(L)$ **then**
 - Acquire $\epsilon(Q)$.
 - $APPEND(Q, L)$.
 - Release $\epsilon(Q)$.
16. $Rg.DUpdate(L)$. □

The procedure $Rg.Find$ is invoked to locate the leaf node which contains a given object O . It has five parameters where P is a node of the tree, O is the object, $lastset$ is the version number of the node from P 's parent, $Bnode$ is the branch node during the downward traversal and $Stime$ is the starting time of the operation. Initially, $Bnode$ is $NULL$, P is the root of the tree and $lastset$ is 0.

$Rg.Find(P, O, lastset, Bnode, Stime)$

1. Acquire $\rho(P)$.
2. **if** $(lastset = NULL)$ or $(ctype(P, lastset, DELETE) = FALSE)$ **then**

- Acquire $\rho(Bnode)$.
 - **if** P is a non-leaf node **then**
 - Let $\{E_i = (MBR(P_i), P_i) \mid i = 1, \dots, k\}$ be the entries in P such that O is inside $MBR(P_i)$.
 - **if** $(k > 0)$ **then**
 - * **if** $(Bnode = NULL)$ and $(k > 1)$ **then**
 - Acquire $\epsilon(Bnode)$.
 - $Bnode := P$.
 - Downgrade $\epsilon(Bnode)$ to $\rho(Bnode)$.
 - else if** $(Bnode = NULL)$ **then**
 - **if** P is in the log table **then** update the version number of P in the log table.
 - **else** insert $(P, version(P))$ into the log table.
 - * For all entries E_i found above, continue the search in parallel invoking $Rg.Find(P_i, O, lastset_i, Bnode, Stime)$ where $lastset_i$ is the version number of entry E_i .
 - Release $\rho(Bnode)$.
 - Release $\rho(P)$.
 - Return.
 - else if** $(Bnode = NULL)$ or $((TIMESTAMP(P) > Stime)$ and $(FTIME(DQ) < Stime))$ **then**
 - **if** $(O$ inside $P)$ **then** return P . **else** release $\rho(P)$.
 - Return.
3. Release $\rho(P)$.
4. $F := Bnode$.
5. Acquire $\rho(Bnode)$.
6. **while** $(F \neq NULL)$ and $(version(F) \neq LOG(F).version)$ **do**
 - $F := GetParent(F)$.
- enddo**

7. **if** ($F = NULL$) **then** $F := \text{Root}$ ⁸
8. Abort all sub-finds underneath F .
9. $Bnode := NULL$.
10. $Stime := \text{Current system time}$.
11. $Rg.Find(F, O, NULL, Bnode, Stime)$. □

The procedure $Rg.DUpdate$ performs the second phase of a delete operation partially. It backtracks and shrinks the MBRs of the nodes until no more shrinkage is needed.

$Rg.DUpdate(L)$

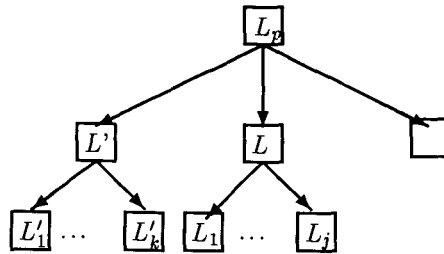
1. $L_p := UGetParent(L)$.
2. **while** ($L_p \neq NULL$)
 - Acquire $\epsilon(L_p)$.
 - Let E_u be the entry in L_p referencing L .
 - Update MBR of E_u and L_p .
 - **if** there is no change for $MBR(L_p)$ **then**
 - Release $\epsilon(L_p)$.
 - Return.
 - Update version number of L_p .
 - Downgrade to $\rho(L_p)$.
 - $L := L_p$.
 - $L_p := UGetParent(L)$.

enddo □

5.4.4 Maintain

The system operation, *maintain*, checks the maintenance queue Q periodically. When it is awakened, it obtains the references to nodes from Q one at a time. Each node is processed according to its property (overflow, underflow). At each maintenance step, at most a subtree consisting of three levels of nodes is locked exclusively. In Figure 5.7, for example, the

⁸The root of the tree.

Figure 5.7: Re-organize the tree at node L

maintain operation is working at L . It first acquires a lock at the parent of L (L_p) to block any other updating operation. It then locks L and its siblings (such as L') to perform the merging of nodes if underflow. If L overflows, then its siblings do not need to be included in the reorganization. The children of L and/or L' (depending on if merging is needed) are locked because of updating their parent pointers. To simplify the algorithm, there is only one *maintain* operation working in the tree. A small modification to the algorithm which changes the maintenance queue management will allow multiple *maintain* operations.

Rg.Maintain()

1. **while** (*TRUE*) **do**

- Sleep for a while.
- Acquire $\rho(Q)$.
- **while** ($Size(Q) > Q_{max}$) **do**
 - Acquire $\epsilon(Q)$.
 - $L := NEXT(Q)$.
 - Release $\epsilon(Q)$.
 - Acquire $\rho(L)$.
 - $L_p := GetParent(L)$.
 - Acquire $\epsilon(L_p)$.
 - Acquire $\epsilon(L)$.
 - $RECT.old := MBR(L_p)$.
 - Update the MBR of L_p with L .
 - **if** *UNDERFL*(L) and (L has no pending update) **then**

```

* Let  $L_p$  has  $k_1$  children including  $L$ .
* Acquire  $\epsilon(L_i)$  for  $i=1,\dots,k_1$ .
* for  $i=1,\dots,k_1$  do
  · Let  $L_i$  has  $k_{ik}$  children.
  · Acquire  $\epsilon(L_{ij})$  for  $j=1,\dots,k_{ik}$ .
endfor
* Re-distribute entries of  $L$  into its siblings.
* Update MBRs of  $L_p$  and  $L_i$ 's.
* Update parental references in  $L_i$ 's and  $L_{ij}$ 's.
* Update version numbers and change types of  $L_p$  and  $L_i$ 's.
* if  $L$  has no entry then mark  $L$  as deleted.
* Release locks on  $L_{ij}$ 's.
* Release locks on  $L_i$ 's.
- if  $\text{OVERFL}(L)$  then
  * Let  $L$  has  $k_2$  children.
  * Acquire  $\epsilon(L_j)$  for  $j=1,\dots,k_2$ .
  * Split  $L$  into  $L$  and  $L'$ .
  * Add a new entry for  $L'$  in  $L_p$ .
  * Update the MBRs of  $L$ ,  $L'$ , and  $L_p$ .
  * Update parental references in  $L$ ,  $L'$  and  $L_j$ 's.
  * Update version numbers, change types of  $L_p$ ,  $L$  and  $L'$ .
  * Release locks on  $L_j$ 's.
  * Release locks on  $L$  and  $L'$ .
- Release  $\epsilon(L_p)$ .
- if  $\text{UNDERFL}(L_p)$  or  $\text{OVERFL}(L_p)$  or  $(\text{RECT.old} \neq \text{MBR}(L_p))$  then
  * Acquire  $\epsilon(Q)$ .
  *  $\text{APPEND}(Q, L_p)$ .
  * Release  $\epsilon(Q)$ .
- Acquire  $\rho(Q)$ .
enddo.
• Release the lock on  $Q$ .

```

enddo.

□

5.4.5 Correctness of the Operations

In this section, we show that there is no deadlock between the operations. We then move on to discuss the correctness of the R-opt tree. In the following discussion, an update operation can either be an insert or a delete operation.

Lemma 5.5 *A user operation can overtake any other user operation.*

Proof: For the three types of user operations, at each step, each operation acquires a lock on a node (P), works on it, and releases the lock before visiting another node (P_n). Before an operation successfully acquires a lock on P_n , a second operation might overtake it by acquiring the lock on P_n first. In the R-opt tree, overtaking between different operations is possible. \square

Theorem 5.7 *The R-opt tree is deadlock free.*

Proof: A deadlock occurs if two operations each holds partial locks and each waits for the other to release its locks. In an R-opt tree, each user operation (search, insert and delete) acquires a single lock on a node at a time. Hence, deadlock cannot occur amongst them. The system operation (maintain) acquires locks on additional nodes, but there is only one maintain operation active in the tree. Therefore, it is obvious that deadlock cannot occur in an R-opt tree. \square

We would like to note that livelock may occur if some user operations are much slower than other operations due to unfortunate timing. For example, a search operation may always be overtaken by update operations. If the version number and the change type of the next visiting node for the search operation change continuously, the search operation will need to be restarted all the time. However, with the similar arguments as presented for the R-tree in Chapter 4, all operations are running at similar priority and they will terminate eventually.

Lemma 5.6 *A search operation will finish eventually.*

Proof: If there are only search operations active in the tree, all search operations will finish their work. Suppose there are some update operations modifying the tree. During the downward traversal of a search operation, it fails to continue when the next node to be visited has disappeared (i.e., removed). However, this is not possible because the mark-and-remove

technique is used for deleted nodes (see procedure *Rg.maintain*). Another situation is where a search operation visits a node and finds a different range from what it knows. In procedure *Rg.Search*, the operation will then purge its results and backtrack to a stable node with its own log table. In both cases, the operation can continue its work. From Theorem 5.7, a search operation cannot be blocked forever. Hence, if there are only infrequent update operations and all operations run at similar priority, the search operation will eventually finish. \square

Lemma 5.7 *An update operation will finish eventually.*

Proof: Each update operation has two phases of work. The first phase is to locate a leaf node to insert or delete an object. This phase is similar to a search operation. As proved in Lemma 5.6, the work will be finished eventually.

The second phase is to backtrack and update the MBRs of nodes. The work cannot continue only when a node has lost its parent or a node is found to have changed during a re-visit. Let U_1 be an update operation in its second phase trying to obtain the parent of node P . Suppose the parent of P is P_p . If P_p has been split or deleted, U_1 will fail to find the parent. However, the parent pointer of P will be adjusted by the maintain operation. Therefore, at the next attempt (see procedure *GetParent*), U_1 will find the correct parent. U_1 releases the lock on P before working on P_p . If P_p is not the correct parent, U_1 will re-acquire a lock on P . At that time, P can be deleted or split. U_1 will then stop because it cannot find the parent of P anymore. The maintain operation, which deleted or split P , will complete the rest of the work for U_1 (see procedure *Rg.maintain*). In all cases, U_1 finishes its work. \square

Similarly to the proofs in other sections, the correctness of the RG protocol depends on the non-overtaking of operations. In the R-opt tree, overtaking is permitted. In the next lemma, we show that when a search operation and an update operation access the same leaf node, they follow a serial order. Before we proceed, we have the following definition.

Effective Timestamp(ET): The *ET* of an insert operation will not change and it is set to the time when it first accesses the root. On the other hand, the *ET* of a search and a delete operation changes. Initially, the *ET* of a search (delete) operation is set to the time it accesses the root. If the operation restarts, the *ET* is reset to the time when it accesses the stable node where it restarts.

Lemma 5.8 *A search operation and an update operation work on a common leaf node according to their ETs.*

Proof: Let there be a search operation (S) and an update operation (U) arriving at leaf node P . Suppose S is younger than U and it acquires a lock on P first (i.e., S overtakes U). Before S starts working on P , it checks the queue DQ (Step 2 in procedure $Rg.search$). U will be found as an entry in DQ . Hence, S restarts and the ET of S is reset. S now becomes older than U . Another situation is where U is younger than S , and U acquires a lock on P first (i.e., U overtakes S). After modifying P , U will set the timestamp of P to its ET. When S arrives at P later and notices P has a younger timestamp, it restarts and obtains a new ET. In the two cases above, the final ET of S will be older than the ET of U . In other words, S can only complete its work at P after U has finished. \square

Lemma 5.9 *In the protocol RG , no search operation will miss an available object with respect to its ET.*

Proof: From Lemma 5.6, a search operation will terminate. Therefore, we only need to consider if a search operation (S) returns all available objects. S will not be restarted if it is not overtaken by a younger update operation and all older update operations have completed their work. From Lemma 5.8, S successfully works on the leaf nodes only when the two conditions are satisfied. Hence, S will see available objects which are inserted/deleted by operations of older ETs. \square

Theorem 5.8 *The protocol RG based on the give-up approach is correct.*

Proof: From Lemma 5.6 and Lemma 5.7, all operations will finish their work. From Lemma 5.9, a search operation will not miss any available object with respect to its ET and there will be no deadlock as proved in Theorem 5.7. Hence, we can conclude that the protocol is correct. \square

5.5 R-Link Tree

In the previous sections, we presented four different extensions to the R-tree to allow concurrent operations. However, recovery of an R-tree from system failures has not been addressed. In this section, we apply the *link technique* proposed by Kung and Lehman[LeY81]

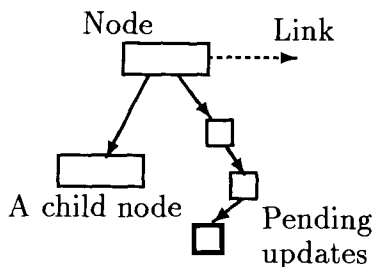


Figure 5.8: A node of an R-link tree.

to modify the R-tree to support recovery after system failures, as well as allowing concurrent operations. We call the new index structure the **R-link tree**. Our objective is to maintain a consistent tree across system crashes. We achieve this by using link pointers and decomposing the user operations into smaller atomic actions (cf. [LoS91]).

The link technique was adapted by Lehman and Yao to B-trees [LeY81]. They modified the B-tree to the *B-link tree* by adding a link pointer to each node. Each node of a B-link tree thus has two types of pointers: its *child pointers* point to the child nodes, and its *link pointer* points to its right neighbor at the same level of the tree. Therefore, link pointers provide an alternate path from a node to each of its child nodes via the leftmost child node. We take advantage of this redundancy for concurrency and recovery.

Like the B-link tree, each node of an R-link tree contains a number of entries, one for each child node, and a link pointer. Using the same idea from the R-opt tree, each node in an R-link tree contains a list of *pending updates* (see Figure 5.8). The pending updates are maintained for the lazy updating of MBR's as a result of *insert's*. When subsequent user operations visit these nodes, the recorded updates are incorporated into the tree level by level, starting at the leaf level. When a user requests an operation, we assign a sequence number, to the operation. This number will identify the operation and all its sub-operations. Each pending update has three fields: the *pending MBR*, the *sequence number* (of the insert operation that caused the change in the MBR), and the identity of the entry referencing the child node on the insertion path. The pending MBR field can be either a MBR or the value *DONE*, the latter indicating the fact that the node's MBR has already been updated and the need to update the parent's MBR. If the identity field has the value of 0, the node is a leaf node.

The search operation traverses the tree with the help of the MBR's and pending updates

	Changes	Levels affected
Insert	Update MBR's	2
	Split a node	1
	Add an entry	2
Delete	Update MBR's	2
	Remove a node	2
Condense	Update MBR's	2
	Remove a node	2
Reorganize	Rearrange nodes	≥ 2

Figure 5.9: Structural changes resulting from an operation

at each visited node. The *insert* operation may *cause* the following changes to the tree: (a) splitting nodes and adding new entries to their parents, and (b) adding and removing pending updates. These two changes may modify the size of the MBR's associated with some nodes. The *delete* operation removes an object and its corresponding entry in a leaf node. It may also cause a node to become empty, flagging the node as "deleted." Besides their main functions, the three user operations also assist in lazy updating, carrying out unfinished structural changes to the R-link tree which are necessitated by earlier *insert*'s and *delete*'s.

The system operation, *condense*, works on at most two levels of nodes in the tree, while the other system operation, *reorganize*, may lock a sub-tree in order to rearrange the nodes in it. In Figure 5.9, we summarize the changes caused by different operations. In order to facilitate the *condense* and the *reorganize* operations, at each level of the R-link tree, we introduce a head pointer pointing to the leftmost node.

Our concurrency control protocol, RL, for R-link trees is implemented by a collection of procedures, for search, insert, and delete operations, as well as those for tree-reorganization. In the following subsections, we discuss them in detail.

Concurrency Control

To allow an orderly access to an R-link tree by user and system operations, we use ρ -locks and ϵ -locks the nodes.

During a *search*, other operations such as *insert* and/or *delete* may be in progress. After a *search* read a child pointer, a concurrent *insert* may have split the child, or a concurrent

condense may have removed the child. In such a case, link pointers provide an alternate path to reach a node or provide information that the child no longer exists. Unlike in a B-tree, however, the nodes at the same level are not linearly ordered according to some key scalar values. A *search* needs to move to the right neighbor, if the current node has been split since the child pointer in the parent was consulted. To check for this condition, we use a pointer pointing to its parent; if there has been a split which has not been reflected in the parent node, then this pointer is set to *NULL*.

Recovery

A recovery scheme for a tree allows the tree to restructure itself into a consistent state after a system failure.

Let us call a state of a B-link tree or an R-link tree *complete* if every node, except its root and the leftmost node at each level, can be accessed both via a child pointer *and* link pointers; otherwise the tree is in an *incomplete* state. As a system failure can occur at any time, a structural change to the tree may be aborted in the middle, leaving the tree in an incomplete state. Shasha and Goodman [ShG88] introduced the concept of the *well-formed* order for a B-tree, which assures that future operations will continue to perform correctly. With the use of the link pointers, Lomet and Salzberg [LoS91] showed that even in an incomplete state, the B-link tree is well-formed. They describe a method whereby, after a system failure, they reconstruct the B-link tree gradually with the atomic actions invoked by normal operations (e.g., *search*).

Similarly, we want to maintain enough information in the R-link tree, so that it can always carry out future operations correctly. When the system crashes, the R-link tree may be left in an incomplete state. There can be a number of structural changes which were under way at the time of the failure. Some operations may have been in the middle of (1) splitting, (2) adding an entry to a node, (3) deleting a node, or (4) updating MBR's. In the second situation, the tree is in an incomplete state, while in all other three situations, the updating of the tree has not been finished.

Each structural change to the tree can be viewed as a sequence of steps and a step can give rise to one of the four situations above. We implement each step as an atomic action, which has the all-or-nothing property. Each action is considered as a database transaction in order to guarantee serializability as well as atomicity. We adopt *write-ahead* logging for all the actions [GrR92]. Each action is first logged in stable storage before it is allowed to

modify the tree. After its successful completion, we write the commit record in the log. With the log, the atomic actions can be restarted and all aborted actions can be rolled back completely. Locks may be held by some transactions when the system fails. We first release all the locks and restart all non-completed operations.

5.5.1 Supporting Procedures

Before describing the atomic actions, we first present two supporting procedures: *GetParent* and *MarkNode*. They are invoked as servicing routines for the atomic actions.

GetParent

The *GetParent(L)* procedure is called by all atomic actions. It returns the parent of the node L . During a system recovery, the parent reference (*ParentPtr*) in L may not have been updated. The *GetParent* first obtains the parent reference in L , accesses the parent and checks if it is the correct parent. If it is not right, it will release the locks and then attempt to retrieve the parent reference later. There is only one parameter for the function. The parameter, L , is a node in the tree which is ρ -locked before the function is called. When *GetParent* finishes its work, the lock on L is released. If L has a parent, then the parent will be ρ -locked.

GetParent(L)

1. **if** ($L = NULL$) **then** return $NULL$.
2. $L_p := PARENT(L)$.
3. Release $\rho(L)$.
4. **if** ($L_p = NULL$) **then** return $NULL$.
5. $\rho(L_p)$.
6. **while** ($L_p \neq NULL$) and (L is not a child of L_p) **do**
 - Release $\rho(L_p)$.
 - Sleep a while.
 - Acquire $\rho(L)$.

- $L_p := \text{PARENT}(L)$.
- **if** $\text{DELETED}(L)$ **then**
 - Acquire $\rho(L_p)$.
 - Return L_p .
- Release $\rho(L)$.
- **if** $(L_p = \text{NULL})$ **then** return NULL .
- Acquire $\rho(L_p)$.

7. **enddo**

8. Return L_p . □

MarkNode

The procedure *MarkNode* is called to record a node as deleted. There are three parameters for the procedure. The first parameter is L which is the node to be marked as deleted and has been ϵ -locked by the invoking procedure. The second parameter, L' , is the right sibling of L . If L' is a new node and has no parent, then the deletion of L is delayed until the update finishes. The third parameter, *lockflag*, is used to indicate if there is any need in acquiring the lock on L' . In the *compress* operations, L' would have acquired the locks before this procedure is called. In the following procedure, DC is a queue to store references of underflowed or deleted nodes.

MarkNode($L, L', \text{lockflag}$)

1. **if** ($\text{lockflag} = \text{TRUE}$) **then** acquire $\rho(L')$.
2. **if** ($L' = \text{NULL}$) or ($\text{PARENT}(L') \neq \text{NULL}$) **then**
 - Mark L as deleted.
 - Acquire $\epsilon(DC)$.
 - $\text{APPEND}(DC, L)$.
 - Release $\epsilon(DC)$.
3. **if** ($\text{lockflag} = \text{TRUE}$) **then** release $\rho(L')$. □

5.5.2 Atomic Actions

While executing any user operation, we perform four different tests on each node visited. We first check if the current node is a *DELETED* node or is underflowed, and if so, invoke *UpdeleteAction* given below. We then check whether the node is *FULL*, and if so, invoke *SplitAction* given below. We also test if the MBR's are up-to-date, and if not, invoke *UpMBRAction* given below. In this test, we make use of the lists of pending updates of P and $PARENT(P)$. If there is a pair of pending updates, U for P and U' for $PARENT(P)$, such that they have the same transaction number and U has the value *DONE* in its *MBR* field, this implies that the MBR of $PARENT(P)$ has not yet been updated. Finally, we test if there is a new entry representing a “new” node to be posted, and if so, we invoke *IndexAction* given below. (A node is new if its parent pointer field has the *NULL* value.) In order to obtain a higher concurrency for the *search* operation, updating actions which do not affect the results of *search* are not included. Only the fourth test is included in each step of a search operation.

Starting with the root, at each node which a user operation visits, we perform the above tests and the corresponding actions depending on the type of operation. Note that there are two structural changes involved in splitting P : the first is to split the node into two nodes, P and P' , and the second is to post a new entry to their parent. We carry out these two changes by two different actions, *SplitAction* and *IndexAction*, respectively (see below). This reduces the rollback effort if the system crashes.

UpDeleteAction

This action is invoked when a node has been marked as deleted or the node is underflowed. It first finds the parent of P and updates the corresponding entry. If the parent's MBR changes as a result, it may necessitate the change of the grandparent's MBR, and so forth, which is carried out by *UpMBRAction*. When the node is underflowed, its reference will be appended to a queue, DC . When the size of DC is more than a preset limit, the *condense* operation will be invoked to re-structure the tree. The action has two parameters, P which is a node in the R-tree and $SeqNo$ which is the transaction number of the current transaction.

UpDeleteAction($P, SeqNo$)

- Acquire $\rho(P)$.

- **if** \neg (*DELETED*(*P*) or *UNDERFL*(*P*)) **then** return.
 - *LP* := *GetParent*(*P*).
 - **if** (*LP* = *NULL*) **then**
 - **if** (*P* = *Root*) and *DELETE*(*P*) **then**
 - * Acquire ϵ (*P*).
 - * *P* := *NULL*.
 - * Downgrade ϵ (*P*) to ρ (*P*).
 - Release ρ (*P*).
 - Return.
 - Acquire ϵ (*LP*).
 - Acquire ϵ (*P*).
 - **if** *DELETE*(*P*) **then**
 - Remove the entry in *LP* which references *P*.
 - **if** *LP* has no entry **then**
 - * *P'* := *NEXT*(*LP*).
 - * *MarkNode*(*LP*, *P'*, *TRUE*).
 - * Return
 - **if** we need to shrink *MBR*(*LP*) **then**
 - * Update the *MBR* of *LP*.
 - * Let *CurPrList* be the list of pending updates in *LP*.
 - * Let *CurPrList.new* be a new element for *CurPrList*.
 - * *CurPrList.new.trans* := *SeqNo*.
 - * *CurPrList.new.MBR* := *DONE*.
 - * *CurPrList.new.index* := 0.
- else**
- Acquire ϵ (*DC*).
 - *APPEND*(*DC*, *P*).
 - Release ϵ (*DC*).

- *Rl.Condense(DC)*.
- Release $\epsilon(P)$.
- Release $\epsilon(LP)$. □

UpMBRAction

The action is used to complete the updates due to the *insert* and *delete* operations. It updates the MBR of $PARENT(P)$, and sets the *MBR* field of U' to *DONE* for split nodes. (See above for the definition of U' .) When the index field of a pending element is 0, then the update is due to a deletion. The MBR updates propagate upwards level by level and it may require several invocations of this action to complete them. There are two parameters for the action. The first parameter, P , is a node in the tree and the second one is the current transaction number.

UpMBRAction(P, SeqNo)

- Acquire $\rho(P)$.
- $LP := GetParent(P)$.
- **if** ($LP \neq NULL$) **then**
 - Acquire $\epsilon(LP)$.
 - Acquire $\epsilon(P)$.
 - Let E_i be the entry in LP referencing P .
 - Let PR_p be the pending update in P whose transaction number is $SeqNo$.
 - **if** ($PR_p.MBR = DONE$) **then**
 - * Remove PR_p .
 - * Let PR_{lp} the pending update in LP whose transaction number is $SeqNo$ and $PR_{lp}.index = i$.
 - * **if** PR_{lp} exists **then**
 - Update the MBRs of LP and E_i
 - $PR_{lp}.mbr := DONE$.
 - else**
 - Update the MBR of LP .

- Let *CurPrList* be the list of pending updates in *LP*.
 - Let *CurPrList.new* be the new element of *CurPrList*.
 - *CurPrList.new.trans* := *SeqNo*.
 - *CurPrList.new.MBR* := *DONE*.
 - *CurPrList.new.index* := 0.
- Release $\epsilon(P)$.
 - Release $\epsilon(LP)$.
- else release $\rho(P)$. □

SplitAction

This action performs only the first half of the structural changes, caused by a split. After creating a new node P' , it moves about half of the entries of P to it. Initially, P' has no parent, which is indicated by its parent pointer, *ParentPtr*, being set to *NULL*. The link pointer of P is changed to point to P' . There is only one parameter, P , which is a node in the tree for the action.

SplitAction(P)

- Acquire $\epsilon(P)$.
- if ($P \neq NULL$) and *FULL*(P) then
 - Acquire $\epsilon(P)$.
 - Let P have k children $\{ P_i \mid i = 1, \dots, k \}$.
 - Acquire $\epsilon(P_i)$ for $i = 1, \dots, k$.
 - Split P into P and P' without re-arranging the order in the entries.
 - *PARENT*(P') := *NULL*.
 - *NEXT*(P') := *NEXT*(P).
 - *NEXT*(P) := P' .
 - Update the parent pointers of P_i 's.
 - Update the MBRs of P and P' .
- Release $\epsilon(P)$. □

IndexAction

It performs the second structural change after a split. It adds a new entry in P 's parent and adjust the parent's MBR accordingly.

IndexAction(P, P')

- Acquire $\rho(P)$.
- **if** ($P = \text{Root}$) **then**
 - Create a new node LP .
 - Acquire $\epsilon(LP)$.
 - Acquire $\epsilon(P)$.
 - Acquire $\epsilon(P')$.
 - Insert two new entries in LP for P and P' .
 - Set new root to LP .
 - Update the MBR of LP .
 - Update the parent pointers of P and P' .
 - Release $\epsilon(P)$.
 - Release $\epsilon(P')$.
 - Release $\epsilon(LP)$.

else

- $LP := \text{GetParent}(P)$.
- Acquire $\epsilon(LP)$.
- Acquire $\epsilon(P')$.
- Insert a new entries in LP for P' .
- Update the MBR of LP .
- Update the parent pointer of P' .
- Release $\epsilon(P')$.
- Release $\epsilon(LP)$.

□

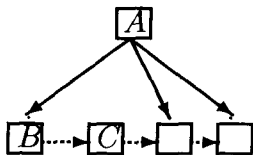


Figure 5.10: Using the link pointer to search.

5.5.3 Search

Like all other user operations, a *search* starts at the root of the R-link tree and descends down to the leaf level. At each visited node (e.g., A in Figure 5.10), a ρ lock on it must be acquired. After it has decided which child nodes to visit next, the ρ lock can be released. If a child node (C in Figure 5.10) is a newly created node by a recent split, it is possible that no entry representing C has been placed in node A . As shown in Figure 5.10, the link pointer at B provides an alternate way to access node C even though A does not yet point to C . In our implementation, when a *search* visits C , it will test the value of the field *ParentPtr* in C . If the field is *NULL*, we start a new sub-search operation to search the subtree rooted at C . If the field is not *NULL* and $MBR(C)$ overlaps with W , a sub-search would have been initiated at A already.

While executing a *search* operation, we perform the *child split test* as stated previously on each visited node. It is because the next node to be visited may have been split. The search operation would need to include the new areas. As the other three tests do not reveal a change in the MBR of the visited node, they are not included in the search. We carry out *search* by calling $Rl.Search(W, P)$, where W is the search window and P is the root of a sub-tree. Initially, we set P to the root R of the tree.

$Rl.Search(W, P)$

1. Acquire $\rho(P)$.
2. **if** P is a non-leaf node **then**
 - Let $\{E_i = (MBR(P_i), P_i) \mid i = 1, \dots, k\}$ be the entries in P such that $(MBR(P_i) \cup U_i) \cap W \neq \phi$, where U_i is the union of the MBR's of the pending updates which point to E_i .
 - **if** $(k > 0)$ **then**

- For all entries E_i found above, continue the search in parallel, invoking $Rl.Search((MBR(P_i) \cup U_i) \cap W, P_i)$.

else for all objects whose MBR's overlap with W return their object ids.

3. **if** ($NEXT(P) \neq NULL$) **then**

- $P' = NEXT(P)$.
- Acquire $\rho(P')$.
- **if** ($PARENT(P') = NULL$) **then**
 - $Rl.Search(W, P')$.
 - $IndexAction(P, P')$.
- Release $\rho(P')$.

4. Release $\rho(P)$. □

5.5.4 Insert

Given an R-link tree with the root R and an object O to be inserted, the *insert* operation adds O to an appropriate leaf in the tree. If the leaf overflows, then it is split, and the *insert* propagates up the tree. The first phase is to select a leaf node to which to add O and insert it. The second phase is to propagate the required updates (i.e., splitting and/or enlarged MBR) upward along the insertion path.

The MBR's of some nodes on the insertion path will change if the object to be inserted covers additional area. If the MBR's of those nodes are updated immediately in the first phase, they may be changed again by a subsequent *condense* or *reorganize* operation, oblivious of the imminent insertion of O . This will cause a subsequent *search* to miss O . We solve this problem by delaying the update, recording the expanded MBR in the list of pending updates, which will be later (in the second phase) used to enlarge the node's MBR. This recording is carried out during the first phase as the *insert* traverses the tree downward along the insertion path. In the mean time, before an expanded MBR replaces the node's MBR, subsequent operations examine both the MBR of the node and its MBR's of the pending updates.

In our algorithm, the *insert* operation attempts to carry out the second phase. The action, $Rl.CleanUpAction$, is used to perform the reorganization of the tree as need. To

simplify its work and to avoid multiple actions working on the same part of the tree, when this action is aborted, it will not be restarted. If the system fails during the updating action, the remaining part is cooperatively carried out by subsequent user operations. An *insert* itself pitches in to incorporate the exiting pending MBR's at some nodes on its insertion path as the nodes' new MBR's. To this end, during the first phase, an *insert* performs the four tests (as described in Section 5.5.2) at each visited node. Depending on the test outcomes, the appropriate atomic actions will be executed to complete tree updates. There is one atomic action to be invoked at the end of phase one: *Rl.InsertObject*, described below, is used only by *insert* to add an object to a leaf node.

More formally, *Insert(O, P, SeqNo)* has four parameters. Initially, it is carried out as follows, where *O* is the object to be added, *P* is the root of the tree, and *SeqNo* is the transaction number.

Rl.Insert(O, P, SeqNo)

- Acquire $\rho(P)$.
- **if** *DELETED(P)* or *UNDERFL(P)* **then**
 - *Rl.Insert(O, PARENT(P), SeqNo)*.
 - Release $\rho(P)$.
 - *UpDeleteAction(P, SeqNo)*.
 - Return.
- **if** *FULL(P)* **then** *SplitAction(P)*.
- Let *CurPrList* be the list of the pending updates of *P*.
- Let $\{L_i \mid i = 1, \dots, l\}$ be the elements in *CurPrList* such that $L_i.MBR = DONE$.
- For all elements L_i found above, invoke *UpMBRAction(P, SeqNo_i)* where *SeqNo_i* is the transaction number associated with L_i .
- **if** *P* is a non-leaf node **then**
 - Find an entry $E_s = (MBR(P_s), P_s)$ in *P*, where $(MBR(P_s) \cup U_s)$ requires the smallest enlargement to include $MBR(O)$ and U_s is the union of the MBR's of pending updates for E_s . Resolve ties by choosing E_s with the smallest $MBR(P_s)$.
 - **If** an enlargement is required in E_s **then**

- * $\epsilon(P)$.
- * Let *CurPrList.new* be the element to be added.⁹
- * *CurPrList.new.trans* := *SeqNo*.
- * *CurPrList.new.MBR* := *MBR(O)*.
- * *CurPrList.new.index* := *s*.
- * Release $\epsilon(P)$.
- *Rl.Insert(O, P, SeqNo)*.
- else *Rl.InsertAction(O, P, SeqNo)*.
- if (*NEXT(P) ≠ NULL*) then
 - $P' := \text{NEXT}(P)$.
 - Acquire $\rho(P')$.
 - if (*PARENT(P') = NULL*) then *IndexAction(P, P')*.
 - Release $\rho(P')$.
- Release $\rho(P)$. □

When the *Rl.InsertObject* is called, it will insert the object *O* into the leaf node *P*. Before the insertion is carried out, it will first verify *P* has not been deleted. After *O* is inserted, it will set up the values of a pending update if updates are needed.

Rl.InsertObject(O, P, SeqNo)

- Acquire $\rho(P)$.
- if *DELETED(P)* then
 - Release $\rho(P)$.
 - *Rl.Insert(O, PARENT(P), SeqNo)*.
 - Return.
- if (*P ≠ NULL*) then
 - $\epsilon(P)$.
 - $\text{MBR}_1 := \text{MBR}(P)$.

⁹The element is created when there is no element with the same set of values.

- Let $L_c = (MBR(O), O)$.
- Insert L_c into P .
- Update MBR of P .
- $MBR_2 := MBR(P)$.
- **if** ($MBR_1 \neq MBR_2$) **then**
 - * Let $CurPrList$ be the list of pending updates in P .
 - * Let $CurPrList.new$ be a new element for $CurPrList$.
 - * $CurPrList.new.trans := SeqNo$.
 - * $CurPrList.new.MBR := DONE$.
 - * $CurPrList.new.index := 0$.
- Release $\epsilon(P)$.
- else** Release $\rho(P)$.

- $Rl.CleanUpAction(L, SeqNo)$. □

The procedure, $Rl.CleanUpAction$, is invoked after an object is added. It will perform the second phase of a normal insert operation to update the tree. When the system fails, it stops and the updating will be left for future operations. It has two parameters. L is the current node it tries to update and $SeqNo$ is the current transaction number.

$Rl.CleanUpAction(L, SeqNo)$

1. **if** $L = NULL$ **then** return.
2. $continue := FALSE$.
3. Acquire $\rho(L)$.
4. $LP := GetParent(L)$.
5. **if** $FULL(L)$ and $(L = Root)$ **then**
 - Create a new root.
 - ϵ -lock the new root.
 - Acquire $\epsilon(L)$.
 - Make it LP .
 - Add L as the single entry in LP .

else

- if $LP \neq NULL$ then acquire $\epsilon(LP)$.
- Acquire $\epsilon(L)$.

6. if $FULL(L)$ then

- Split L into L and L' .
- $NEXT(L') := NEXT(L)$.
- $NEXT(L) := L'$.
- $PARENT(L') := NULL$.
- $continue := TRUE$.
- Add L' to LP .
- if L has k children, L_1, \dots, L_k then
 - Acquire $\epsilon(L_i)$ for $i = 1, \dots, k$.
 - Update the parent pointer in L_i for $i = 1, \dots, k$.
 - Release $\epsilon(L_i)$ for $i = 1, \dots, k$.

7. if $LP \neq NULL$ then

- Let L be the i^{th} child of LP .
- Let L_p and LP_p be the pending elements in L and LP correspondingly where $L_p.trans = SeqNo = LP_p.trans$, $LP_p.index=i$ and $L_p.MBR = DONE$.
- if LP_p and L_p exist then
 - $continue := TRUE$.
 - Update the MBR of LP with $LP_p.MBR$.
 - $LP_p.MBR := DONE$.
 - Remove L_p .

8. Release $\epsilon(L)$.

9. if $LP \neq NULL$ then

- Release $\epsilon(LP)$.
- if $continue$ then $Rl.CleanUpAction(LP, SeqNo)$.

□

5.5.5 Delete

We use two phases to implement the *delete* operation. The first phase is for locating the leaf node L which points to the object O to be deleted. As in the *search* operation, at each visited node, a *delete* operation may branch off into multiple sub-delete operations, which is similar to the *search* operation. There may be multiple descending paths traversed before the leaf node, which contains the object to be removed, can be located. In the first phase, only ρ locks need to be held on at most two consecutive non-leaf nodes at a time. In the second phase, we need to perform the required updates, avoiding interference with the concurrent *insert*'s and the changing parent pointers and MBR's.

We use the “mark-and-remove” approach. If a node becomes empty as a result of deleting O from L , then it is flagged as “deleted.” In the first phase, the node is not removed immediately, because there can be other concurrent operations working on it. “Deleted” nodes are later garbage-collected periodically. If a node underflows after a deletion, we append it to a queue, DC , which is used to record the underflowed nodes. If its size reaches a threshold ($Max_{underfl}$), the *condense* operation (see Section 5.5.7) is invoked to perform the delayed updates.

While descending the tree during the first phase of a *delete* operation, we perform the four tests we described earlier on each visited node. Atomic actions will be executed to complete tree updates, including those necessitated by other *delete*'s. The atomic action, *Rl.DAction*, is used to delete the unwanted object and to flag the leaf node as necessary.

There are three arguments for the *Rl.Delete*. O is the object to be deleted, P is the root of the tree initially and $SeqNo$ is the transaction number of the current delete operation.

Rl.Delete($O, P, SeqNo$)

1. Acquire $\rho(P)$.
2. **if** *DELETED*(P) **then**
 - *Rl.Delete*($W, PARENT(P), SeqNo$)
 - Release $\rho(P)$.
 - *UpDeleteAction*(P).
 - Return.
3. **if** *FULL*(P) **then** *SplitAction*(P).

4. Let *CurPrList* be the list of pending updates at *P*.
5. Let $\{L_i \mid i = 1, \dots, l\}$ be the elements in *CurPrList* such that $L_i.MBR = DONE$.
6. For all elements L_i found above, invoke $UpMBRAction(P, SeqNo_i)$ where $SeqNo_i$ is the transaction number associated with L_i .
7. **if** *P* is a non-leaf node **then**
 - Let $\{E_i = (MBR(P_i), P_i) \mid i = 1, \dots, k\}$ be the entries in *P* such that $(MBR(P_i) \cup U_i) \cap W \neq \phi$, where U_i is the union of the MBR's of the pending updates which point to E_i .
 - **if** ($k > 0$) **then**
 - For all entries E_i found above, continue the search in parallel invoking $Rl.Delete(O, P_i, SeqNo)$.
- else**
 - Find an entry P_c in *P* where $P_c = (MBR(O), O)$.
 - **if** P_c exists **then** $DAction(O, P, SeqNo)$.
8. $P' := NEXT(P)$.
9. **if** ($P' \neq NULL$) and ($PARENT(P') = NULL$) **then**
 - Continue with $Rl.Delete(O, P', SeqNo)$.
 - $IndexAction(P, P')$.
10. Release $\rho(P)$. □

The action, $DAction$, is used to remove an object from the tree and sets up a pending update for the updating if needed. It has three parameters. O is the object to be removed and P is the leaf node containing the reference to the object. $SeqNo$ is the current transaction number.

$DAction(O, P, SeqNo)$

1. Acquire $\epsilon(P)$.
2. **if** ($DELETED(P)$ or ($P = NULL$)) **then** return.

3. **if** there is an E_d in P such that $E_d = (MBR(O), O)$ **then**

- Remove entry E_d from P .
- Update the MBR of P .

else if $MBR(P)$ has been shrunk **then**

- Let $CurPrList$ be the list of pending updates in P .
- Let $CurPrList.new$ be a new element for $CurPrList$.
- $CurPrList.new.trans := SeqNo$.
- $CurPrList.new.MBR := DONE$.
- $CurPrList.new.index := 0$.

4. **if** P has less than MIN_{allow} entries **then** $UpDeleteAction(P, SeqNo)$.

5. Release $\epsilon(P)$. □

5.5.6 Compress

As commented previously, we use two system operations, *condense* and *reorganize*. They are used to improve the search performance in an R-link tree by minimizing the overlapping and dead spaces between MBR's, and have a balanced tree. The tree is reorganized periodically by these two operations.

Condense

The *condense* operation is invoked if the number of nodes in DC exceeds the threshold, $Max_{underfl}$. The operation retrieves nodes from DC one at a time. It will first verify if the node is still underflowed before merging it to improve storage utilization. *condense* starts at the leaf level and works its way upward. At each level, it accesses the leftmost node by means of a header pointer, which we mentioned earlier, and traverse the level horizontally. When an underflowed node is located, its entries are merged with those of some of its siblings. MBR's are updated and link pointers are adjusted as necessary. If the node becomes empty, it is marked as "deleted." At each step, an underflowed node, say P , moves its entries to both its right and left siblings or just one of them depending on the position of P . If a sibling of P has a different parent than P , updating of the MBR's for different parents are needed. To maintain the simplicity of the operation, only siblings have the same parent as

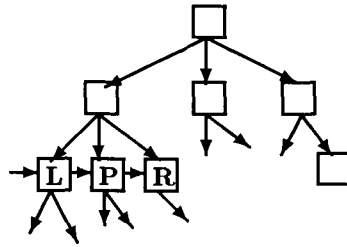


Figure 5.11: Collapsing nodes in a condense operation.

P are considered. In Figure 5.11, *condense* is merging the three nodes labeled **L**, **P** and **R**. Note that, after the merge, the ordering of link pointers among the child nodes of **L** and **R** will remain the same. The link pointer between **L** and **P**, and between **P** and **R** are of course removed.

At each step, the operation works on at most three nodes of the same parent and their child nodes at the same time. All these nodes are locked to avoid potential interference from other operations. The locks are acquired in the same order as in the *insert operation*, as explained before, to avoid potential deadlock.

condense does not attempt to reduce the dead space or overlapping areas. To do so would require rearranging entries within a node, which in turn would cause rearrangement of links among its child nodes. As commented earlier, the overhead required would be excessive. Only one condense operation is allowed to be active at a time. This will reduce the effort to re-structure the under-utilized R-tree. There is only one parameter, DC which contains the references of the underflowed nodes, for the operation. In the procedure, $Max_{underfl}$ is a constant whose value of the maximal limit of the number of marked nodes before reorganization and H_h is the head link pointer at level h of the tree.

Rl.Collapse(DC)

1. **if** ($H_h = NULL$) or ($size(DC)^{10} < Max_{underfl}$) **then** return.
2. Acquire $\epsilon(DC)$.
3. Let node P be the node referenced by H_h .
4. $DC' := DC$.

¹⁰It returns the size (the number of references) in queue DC .

5. Empty DC .
6. Release $\epsilon(DC)$.
7. Obtain a new transaction number, $SeqNo$.
8. $CondAction(NULL, P, h, DC', SeqNo)$. □

Each step of the *condense* operation is performed by the *CondAction* action. It has five parameters. The action works on the current node, P , $LeftP$ is the left sibling of P and h is the current level of the tree. Initially, h equals to the height of the tree. It is used to obtain the head pointer of the link pointers at each level. At level i , the head pointer is represented by H_i . The forth parameter DC is a queue containing the references of the underflowed nodes and $SeqNo$ is the current transaction number.

$CondAction(LeftP, P, h, DC, SeqNo)$

1. **if** ($h = 0$) or ($P = NULL$) or ($size(DC) = 0$) **then** return.
2. **if** ($LeftP \neq NULL$) **then**
 - Acquire $\epsilon(LeftP)$.
 - Let P_i 's be children of $LeftP$ where $i = 1, \dots, x$.
3. Acquire $\epsilon(P)$.
4. Let P_j 's be children of P where $j = 1, \dots, y$.
5. **if** ($NEXT(P) \neq NULL$) **then**
 - Acquire $\epsilon(NEXT(P))$.
 - Let P_k 's be children of $NEXT(P)$ where $k = 1, \dots, z$.
6. **if** P is in DC **then**
 - **if** (P is not a leaf node) **then**
 - **if** ($LeftP \neq NULL$) **then** acquire $\epsilon(P_i)$ where $i = 1, \dots, x$.
 - Acquire $\epsilon(P_j)$ where $j = 1, \dots, y$.
 - **if** ($NEXT(P) \neq NULL$) **then** acquire $\epsilon(P_k)$ where $k = 1, \dots, z$.
 - Move entries P_j 's to $LeftP$ and/or $NEXT(P)$.
 - **if** (P is not a leaf node) **then**

- Adjust link pointers of P_j 's, P_i 's, and P_k 's.
- Update parent references of P_j 's.
- Release $\epsilon(P_i)$ where $i = 1, \dots, x$.
- Release $\epsilon(P_j)$ where $j = 1, \dots, y$.
- Release $\epsilon(P_k)$ where $k = 1, \dots, z$.
- **if** P is empty **then** $\text{MarkNode}(P, \text{NEXT}(P), \text{FALSE})$.
- Remove the reference of P from DC .
- Update the MBR of $\text{Left}P$.
- Update the MBR of $\text{NEXT}(P)$.
- **if** ($\text{MBR}(\text{Left}P)$ has changed) **then**
 - Let CurPrList be the list of pending updates in $\text{Left}P$.
 - Let CurPrList.new be a new element for CurPrList .
 - $\text{CurPrList.new.trans} := \text{SeqNo}$.
 - $\text{CurPrList.new.MBR} := \text{DONE}$.
 - $\text{CurPrList.new.index} := 0$.
- **if** ($\text{MBR}(\text{NEXT}(P))$ has changed) **then**
 - Let CurPrList be the list of pending updates in $\text{NEXT}(P)$.
 - Let CurPrList.new be a new element for CurPrList .
 - $\text{CurPrList.new.trans} := \text{SeqNo}$.
 - $\text{CurPrList.new.MBR} := \text{DONE}$.
 - $\text{CurPrList.new.index} := 0$.

7. if ($\text{NEXT}(P) \neq \text{NULL}$) **then**

- Release $\epsilon(\text{Left}P)$.
- $\text{Left}P := P$.
- $P := \text{NEXT}(P)$.
- Release $\epsilon(\text{Left}P)$ and $\epsilon(P)$.
- $\text{CondAction}(\text{Left}P, P, h, DC, \text{SeqNo})$.

else

- Release $\epsilon(\text{Left}P)$ and $\epsilon(P)$.

- $h := h - 1$.
- $P := H_h$.
- $CondAction(NULL, P, h, DC, SeqNo)$. □

Local Reorganization

The *reorganize* operation tries to restructure the tree to achieve better search performance. It is an expensive operation which should not be invoked frequently. There are a number of possible ways to improve the search performance on an R-link tree, including:

- Cutting down the dead space in the MBR's associated with the nodes; this will reduce unnecessary searches.
- Reducing overlapping among the MBR's of the child nodes of each node; this will cut down the number of sub-searches.
- Clustering "near" objects; sub-searches will branch out only at a low level of the tree.

Depending on different object sizes and distributions, we can set up different performance criteria for nodes. We call a node **under-performing** if it does not satisfy the criteria. We use a queue, *DQ*, to keep track of the under-performing nodes. For this purpose, we add a new field in each node to contain a search performance measure. During the first phase of *insert* and *delete*, or the descending phase of *search*, each visited node is tested against the performance criteria. The nodes that fail the test are added to *DQ*. When the size of *DQ* exceeds a threshold, we invoke the *reorganize* operation. The operation retrieves the nodes from *DQ* for the reorganization.

There are two parameters for *Rl.reorganize*. The first parameter is *DQ* which is the queue containing references to nodes whose search performance levels are poor. The second one, *SeqNo*, is a transaction number assigned to the operation. In Figure 5.12, we show the nodes used in a step of local reorganization.

Rl.reorganize(DQ, SeqNo)

1. Acquire $\epsilon(DQ)$.
2. **if** ($DQ = NULL$) **then** return.
3. $P := NEXT(DQ)$.

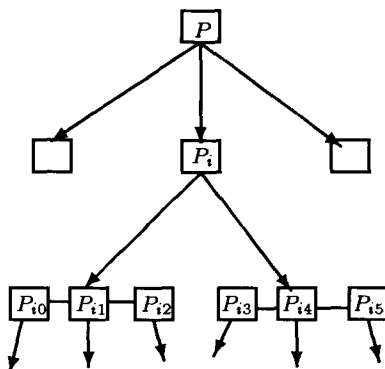


Figure 5.12: Local reorganization.

4. if $P \neq NULL$ then

- Acquire $\rho(P)$.
- Let there be k children of P .
- if $k > 0$ then
 - Acquire $\rho(P_i)$ where $i = 1, \dots, k$.
 - Let P_1 and P_4 be nodes which have entries to be re-arranged and neither is at the border of of the subtree headed at P .
 - **while** such a pair exists **do**
 - * ρ lock the subtrees rooted at P_1 and P_4 .
 - * ρ lock the subtree rooted at P_0 which is the left neighbor of P_1 .
 - * ρ lock the subtree rooted at P_2 which is the right neighbor of P_1 .
 - * ρ lock the subtree rooted at P_3 which is the left neighbor of P_4 .
 - * ρ lock the subtree rooted at P_5 which is the right neighbor of P_4 .
 - * During the acquisition of the share locks, replicate the 6 subtrees and let them be T_0, T_1, \dots, T_5 .
 - * Remove all link pointers in T_1 and T_4 .
 - * Re-organize the entries in T_1 and T_4 .
 - * Update the link pointers of $T_0, T_1, T_2, T_3, T_4, T_5$.
 - * Update the parent pointers and MBRs in T_1 and T_4 .
 - * $\epsilon(P)$.

- * ϵ lock the subtrees rooted at P_1 and P_4 .
 - * ϵ lock the subtree rooted at P_0, P_2, P_3 and P_5 .
 - * Replace subtrees rooted at P_l with T_l for $l=0, \dots, 5$.
 - * Update MBRs of P and P_i .
 - * Release all ϵ -locks in subtrees rooted at P_l for $l=0, \dots, 5$. For the nodes, P_l 's, downgrade their locks to ρ -locks.
 - * $\rho(P)$.
 - * Let P_1 and P_4 be the next pair of nodes which have entries to be rearranged.
- end while**
- Release $\rho(P_i)$ for $i=1, \dots, k$.
 - Release $\rho(P)$.
 - Remove reference of P from DQ .
 - **if** $MBR(P)$ has changed **then**
 - * Let $CurPrList$ be the list of pending updates in P .
 - * Let $CurPrList.new$ be a new element for $CurPrList$.
 - * $CurPrList.new.trans := SeqNo$.
 - * $CurPrList.new.MBR := DONE$.
 - * $CurPrList.new.index := 0$.

5. Release $\epsilon(DQ)$.

6. $Rl.reorganize(DQ, SeqNo)$. □

5.5.7 Serializability and Multiple Searches

In the R-link tree, there are two possible ways to solve the serialization problem which is described in Section 4.1.5. The first method is to avoid overtaking of operations which has been adopted in the previous index structures. This can be achieved by storing transaction numbers in the nodes of the tree. When an operation visits a node, it checks if its preceding operation has finished with the node; otherwise it will quit and re-visit the node later. However, with this pessimistic method, when an operation is aborted unexpectedly, the subsequent operations will starve and cannot continue. We believe that a better solution is to assume that such scenarios are infrequent and to validate a transaction after it has finished its operation but before it commits. However, if we abort an *insert*, a lot of recovery work

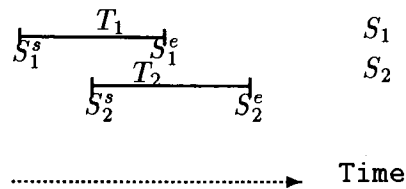


Figure 5.13: Overlapping search times.

would be required. Therefore, we abort only *search*'s to achieve consistency in the results. We maintain a table, which, for each *search* operation S , records its starting time (S^s) and ending time (S^e). Each inserted object has a timestamp to show when it was inserted. The timestamp can be viewed as the finishing time of an *insert* operation. Old entries in the tables for by *search* operations are purged periodically.

If there are two *search* operations, S_1 and S_2 , and the following conditions are all satisfied, then either S_1 or S_2 should be aborted and restarted (see Figure 5.13).

1. Operational times T_1 of S_1 and T_2 of S_2 overlap.
2. The search windows of S_1 and S_2 intersect, i.e., $W_1 \cap W_2 \neq \phi$ where W_1 and W_2 are the search windows of S_1 and S_2 , respectively.
3. There is more than one object added/deleted inside $W_1 \cap W_2$ and the timestamps of the updates are within the overlap of T_1 and T_2 .

No major changes are needed to the algorithms presented in the previous sections. Before a *search* commits and returns its result, it will need to check the above conditions. The check can be done in the order as shown. If the test fails, then no further test will be needed. If all three conditions are satisfied, then the *search* operation is restarted.

5.6 Quad-R Tree

In this section, we present the **quad-R tree**. A quad-R tree is a hierarchical representation for rectangular data. Its structure is similarly to the quad-B tree in the previous chapter. Each entry in a leaf node represents a corresponding terminal quadrant of a quadtree. However, only rectangles that are enclosed minimally by the quadrant can be referenced via

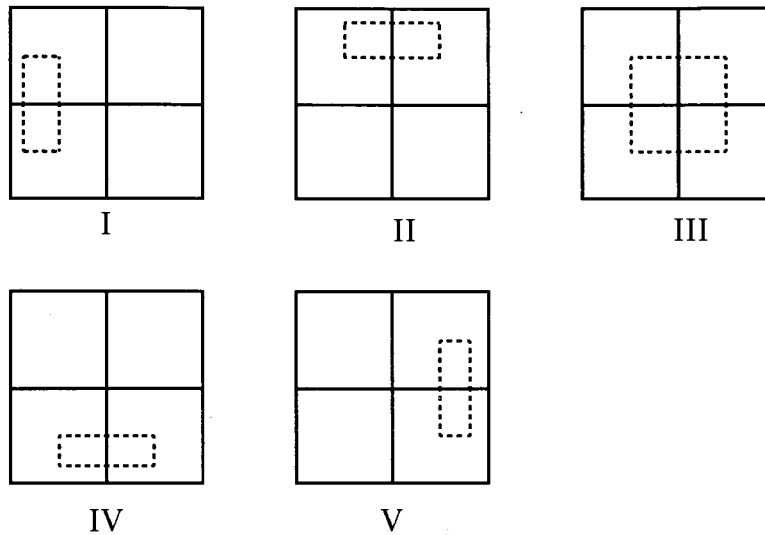


Figure 5.14: Five types of rectangles within a quadrant.

the entry. With the encoding scheme, the quadrants can be ordered according to their sizes and locations in the data space.

The quad-R tree is a B^+ -tree. Each entry of a node is of the form $(Value, P)$ where P is a pointer to a child node, and $Value$ is a scalar value which is the largest scalar value amongst the entries in P . At each level, the rightmost entry of the rightmost node has the largest possible value in the subtree. A leaf node contains entries of the form $(Value, B)$ where B is a pointer referencing a bucket, and $Value$ is the corresponding scalar value for the terminal quadrant. In each terminal quadrant, it can store five different types of rectangular objects as shown in Figure 5.14.

In a quad-R tree, we use the same encoding scheme as in the quad-B tree in the previous chapter. The key for a quadrant at level d (where $l > 0$) is encoded as follows. Let D be the largest depth of any terminal quadrant and the entire data space is encoded as 0. Let the quadrant q have the key value k and its parent q_p have the key value k' . Then

$$k = k' + s5^{D-d} \quad (5.1)$$

where

- $s = 1$, if q is the northwest corner of q_p
- $s = 2$, if q is the southwest corner of q_p

- s = 3, if q is the southeast corner of q_p
- s = 4, if q is the northeast corner of q_p

For the following presentation of the algorithms, we represent the transformation by $\varphi(R)$ where R is either a search window or the MBR of an object.

Similarly to the R-link tree, we have adopted the *link technique* described in [LeY81] to support concurrent operations on the quad-R tree. With this technique, operations are allowed to release the lock on a node they hold before they obtain a new lock on the next node. However, it requires the addition of new edges (*link pointers*) to the tree in order to avoid anomalies.

Our concurrency control protocol, QR, for quad-R trees is implemented by a collection of procedures, for search, insert, and delete operations, as well as those for tree-reorganization. In the following sections, we discuss them in detail.

5.6.1 Search

To perform a search operation, we use range queries instead of applying the given search window directly. A given search window, W , is divided into different regions. Each region will overlap with a number of quadrants which can be at the same or different levels of a corresponding quadtree. In Figure 5.15, the search window W generates 4 ranges as $[0, 0]$, $[1000, 1000]$, $[1300, 1444]$ and $[4000, 4444]$ if there are 4 levels in the corresponding quadtree.

During the calculation of the ranges, we try to have a small number of short ranges in order to have good search performance. In our method, we divide the search window continuously until the total length of the ranges cannot be shortened. We first find out the smallest quadrant which encloses the window. We then use the center lines of the quadrant to divide the window into smaller regions. Each region can be represented by a range from the minimum and the maximum scalar values of the corresponding quadrants it overlaps. A region is further divided into sub-regions if the total length of the new ranges is shorter than the range before. The procedure is repeated for all regions. At the end, the window search query is transformed into a number of range queries. To perform a search operation, we call the procedure *Qr.Search* to compute the ranges and invoke the range query procedure afterwards. Formally, the procedure *Qr.Search* has two parameters where W is the search window and R is the root of the tree. In the procedure, *LeafStack* is a stack which is used

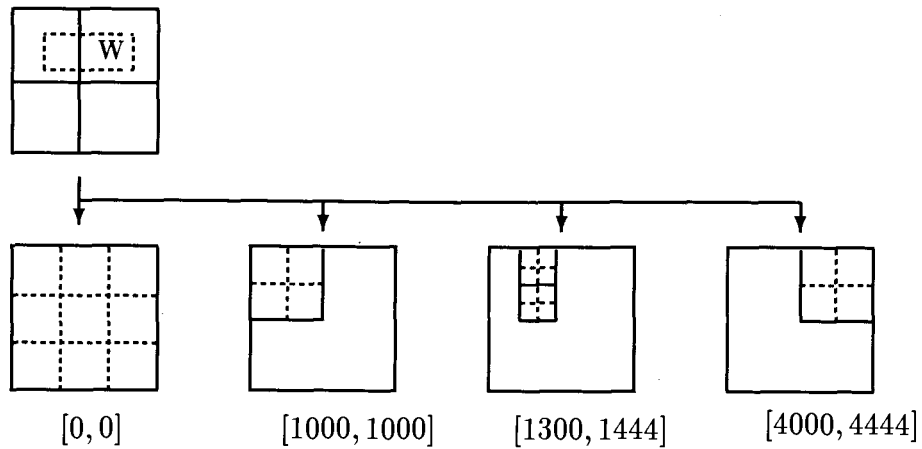


Figure 5.15: A search window divided into 4 query ranges.

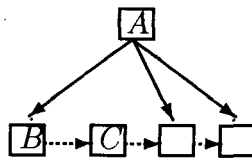


Figure 5.16: Using the link pointer to search.

to store the references of all the leaf nodes visited by the search operation.

$Qr.Search(W, R)$

1. Calculate the set of ranges, $\{RangeSet\}$, for W .
 2. Let there be k ranges as $RangeSet_1, \dots, RangeSet_k$.
 3. $LeafStack := NULL$.
 4. **if** ($k > 0$) **then**
 - **for** $i = 1, \dots, k$ **do**
 - Let $RangeSet_i$ be represented by $[Low_i, High_i]$.
 - $OldValue := \infty$.
 - $Qr.SearchW(W, R, Low_i, High_i, OldValue, LeafStack)$.
- enddo**

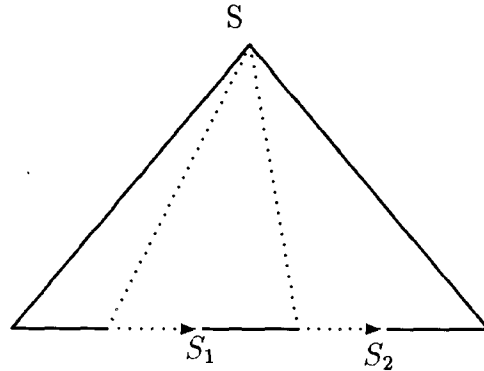


Figure 5.17: Performing a search.

- **if** ($LeafStack \neq NULL$) **then**

- Release all ρ -locks on nodes which are referenced by $LeafStack$.
- $FREE(LeafStack)$. □

The procedure $Qr.SearchW$ carries out the work of each range query. It uses Low to descend from the root to a leaf node. At each step, it releases its lock before visiting another node in the tree. Let S_1 be a range query visiting node A as shown in Figure 5.16, a ρ lock on it is first acquired. After it has decided which child node to visit next, the ρ lock is released. When it visits node B , B may have been split and the right sibling of B (C) is a new node whose corresponding entries have not have been put in A . When this happens, the link pointer at B then provides an alternate way to access node C . In our implementation, when a *search* visits B , it will verify the scalar value of B in A against the largest scalar value of B . If they are different, a split must have been occurred, and a new sub-search operation will be created to search the subtree rooted at C .

When a search operation reaches a leaf node, it holds the lock on the node and traverses along the link pointers until it reaches a leaf node whose first entry has a scalar value larger than the range of the search. In Figure 5.17, the search operation S generates two range queries S_1 and S_2 . S_1 is executed first. When S_1 reaches the leaf level, it uses a stack ($LeafStack$) to record the leaf nodes it visits. The ρ -locks acquired at the leaf nodes by S_1 are not released. After S_1 has finished its work, S_2 starts its traversal similarly to S_1 but with a different range. After all range queries have been completed, $LeafStack$ is used to release all the ρ -locks on the leaf nodes visited by S . For each invocation of

$Qr.SearchW(W, R, SValue, High, OldValue, LeafStack)$, W is the search window, R is a node of the tree, $SValue$ and $High$ are the low and the high value of the range, respectively, $OldValue$ is the largest number in the node visited previously, and $LeafStack$ is the stack contains the references of leaf nodes. Initially, $OldValue$ is the value of the largest possible number, and $LeafStack$ is empty.

$Qr.SearchW(W, R, SValue, High, OldValue, LeafStack)$

1. Acquire $\rho(R)$.
2. **if** $DELETED(R)$ **then** $R := FindChild(HeadPtr, SValue)$ ¹¹.
3. **if** R is a non-leaf node **then**
 - Let $\{E_i = (R_i, V_i) \mid i = 1, \dots, k\}$ be the entries in R such that $V_{i-1} < SValue \leq V_i$.
 - **if** $(SValue > V_n)$ and $(NEXT(R) \neq NULL)$ **then**
 - $R' := NEXT(R)$.
 - $Qr.SearchW(W, R', SValue, High, OldValue, LeafStack)$.
 - else** $Qr.SearchW(W, R_i, SValue, High, V_i, LeafStack)$.
 - Release $\rho(R)$.
- else**
 - Return all objects in R inside W .
 - $PUSH(R, LeafStack)$.
 - $R := NEXT(R)$.
 - Acquire $\rho(R)$.
 - Let R has n entries.
 - **while** $(R \neq NULL)$ and $(V_1 \leq High)$ **do**
 - Return all objects in R inside W .
 - $PUSH(R, LeafStack)$.
 - $R := NEXT(R)$.

¹¹ $HeadPtr$ is the head link pointer at the current level.

- Acquire $\rho(R)$.

enddo

- **if** ($R \neq \text{NULL}$) **then** release $\rho(R)$. □

The *FindChild* is invoked to find the correct node to visit when an intended visiting node is deleted. Initially, the procedure starts at the leftmost link (the head link pointer) at the current level.

FindChild(L, SValue)

1. **if** ($L = \text{NULL}$) **then** return *NULL*.

2. Acquire $\rho(L)$.

3. $Next := \text{NEXT}(L)$.

4. Acquire $\rho(Next)$.

5. Let V_1, \dots, V_n be the values of the entries in *Next*.

6. **while** ($SValue > V_1$) **do**

- Release $\rho(L)$.

- $L := Next$.

- $Next := \text{NEXT}(L)$.

- **if** ($Next = \text{NULL}$) **then** return *L*.

- Acquire $\rho(Next)$.

- Let V_1, \dots, V_n be the values of the entries in *Next*.

enddo

7. Release $\rho(Next)$.

8. Return *L*. □

5.6.2 Insert

In a quad-R tree, when a node is split, a link pointer is created to *link* the two nodes. The *link* is important because it routes other operations to the missing part of a split node. There are two steps in splitting. The first step establishes the link pointer and the second step updates the parent of the split nodes. In order to backtrack in the second phase, we use an additional variable, *ParentPtr*, at all nodes in the tree. The *ParentPtr* of node *P* is a reference pointing to the parent node of *P*. Together with the link pointers, they provide accesses to parents of all nodes. In the second phase, when a node is split, at most a subtree of three levels will be exclusively locked. The *ParentPtr*'s and link pointers are adjusted for the nodes in each split.

The work in the first phase is carried out by the procedure *Qr.Select* which locates the leaf node and the second phase is done by the procedure *Qr.CleanUp*. More formally, *Insert(O, R)* has two parameters where, *O* is the object to be added, and *R* is the root of the tree initially.

Qr.Insert(O, R)

1. $SValue := \varphi(O)$.
2. $L := Qr.Select(O, SValue, R)$.
3. Upgrade $\rho(L)$ to $\epsilon(L)$.
4. **if** there is an entry E_O whose associated value is $SValue$ **then**
 - Let B be the bucket associated with E_O .
 - Add O to B .

else

- Create entry E_O whose associated value is $SValue$.
- Add O to B which is the bucket referenced by E_O .
- Add E_O to node L .
- Downgrade $\epsilon(L)$ to $\rho(L)$.
- *Qr.CleanUp(L, SValue)*.

□

Qr.Select(O, SValue, R)

1. Acquire $\rho(R)$.
2. **if** ($DELETED(R)$) **then** $R := FindChild(HeadPtr, SValue)$.
3. Let R has n entries.
4. **if** $SValue > V_n$ **then**
 - $Qr.Select(O, SValue, NEXT(R))$.
 - Release $\rho(R)$.
 - Return.
5. **if** R is a non-leaf node **then**
 - Let $E_s = (R_s, V_s)$ be the entry in R such that $V_{s-1} < SValue \leq V_s$.
 - Release $\rho(R)$.
 - $Qr.Select(O, SValue, R_s)$.

else return R . □

The procedure $Qr.CleanUp$ is used in the second phase of an insert operation. At each level during updating, it invokes procedure $GetParent$ to obtain the correct parent of the current node before splitting or adding a new entry. Because the lock on L is released while finding the correct parent, the procedure needs to confirm that L is still overflowed before performing the split.

$Qr.CleanUp(L, SValue)$

1. **if** $L = NULL$ **then** return.
2. Acquire $\rho(L)$.
3. **if** $OVERFL(L)$ **then**
 - **if** $L = Root$ **then**
 - Create a new root,
 - ϵ -lock the new root,
 - Add L as the child of $Parent$.
 - Make it $Parent$.

```

    else  $Parent := GetParent(L, SValue)$ .
    • Acquire  $\epsilon(Parent)$ .
    • Acquire  $\epsilon(L)$ .
    • if  $OVERFL(L)$  then
        – Let  $L_1, \dots, L_n$  be children of  $L$ .
        – Acquire  $\epsilon(L_i)$ 's for  $i = 1, \dots, n$ .
        – Split  $L$  into  $L$  and  $L'$ .
        – Add  $L'$  to  $Parent$ .
        – Update the parent pointer of  $L_i$ 's.
        – Update the link and parent pointers of  $L$  and  $L'$ .
        – Release  $\epsilon(L_i)$ 's for  $i = 1, \dots, n$ .
    • Release  $\epsilon(L)$ .
    • Downgrade  $\epsilon(Parent)$  to  $\rho(Parent)$ .
    •  $Qr.CleanUp(Parent, SValue)$ .

else release  $\rho(L)$ . □

```

The procedure *GetParent* is called to return the parent of the node L . Because of a previous splitting or merging of nodes, the parent reference (*ParentPtr*) in L may not have been updated. The *GetParent* first obtains the parent reference in L , accesses the parent and checks if it is still the right parent. If it is not, the link pointer will be used to trace for the right one. The procedure may be running into a chasing loop if the parent of L is kept changing. There are two parameters for the procedure. The first parameter, L , is a node in the tree which is ρ -locked before the function is called. The second parameter, $SValue$, is used to decide when the search should finish. When *GetParent* finishes its work, the lock on L is released. If L has a parent, the parent will be ρ -locked after the call.

GetParent($L, SValue$)

1. **if** ($L = NULL$) **then** return $NULL$.
2. $L_p := PARENT(L)$.
3. Release $\rho(L)$.

4. **if** ($L_p = NULL$) **then** return $NULL$.
 5. Acquire $\rho(L_p)$.
 6. Let V_1, \dots, V_n be the values of the entries in L_p .
 7. **if** ($SValue < V_1$) and (L is not a child of L_p) **then**
 - Release $\rho(L_p)$.
 - $L_p := \text{head of link pointer at the level}$.
 - Acquire $\rho(L_p)$.
 8. **while** ($SValue > V_n$) and (L is not a child of L_p) **do**
 - $Next := NEXT(L_p)$.
 - Release $\rho(L_p)$.
 - **if** ($L_p = NULL$) **then** return $NULL$.
 - $L_p := Next$.
 - Acquire $\rho(L_p)$.
 - Let V_1, \dots, V_n be the values of the entries in L_p .
- enddo**
9. Return L_p . □

5.6.3 Delete

There are two phases to implement the *delete* operation. The first phase is to locate the leaf node L which contains the rectangular object O and deletes it afterwards. In the second phase, we need to perform the required updates while avoiding interference with other updating operations.

We use the “mark-and-remove” approach. If a node becomes empty as a result of deleting O from L , then it is flagged as “deleted.” “Deleted” nodes are later garbage-collected periodically because there can be other concurrent operations working on it. The first phase is carried out by the procedure *Qr.Select*. It locates the leaf node which contains the unwanted object. The second phase is done by *Qr.Condense* which reorganize the tree

afterwards. There are two arguments for the *Qr.Delete* where O is the object to be deleted and R is the root of the tree initially.

Qr.Delete(O, R)

1. $SValue := \varphi(O)$.
2. $L := Qr.Select(O, SValue, R)$.
3. **if** $L \neq NULL$ **then**
 - Let E_O be an entry in L whose associated value is $SValue$.
 - Let B be the bucket associated with E_O .
 - Remove O from B .
 - Remove entry E_O in node L if B is empty now.
 - Downgrade $\epsilon(L)$ to $\rho(L)$.
4. Release $\rho(L)$.
5. *Qr.Condense*(L). □

The procedure *Qr.Condense* is used in the second phase of a delete operation. At each level during updating, it invokes procedure *GetParent* to obtain the correct parent of the current node before splitting or adding a new entry. Because the lock on L is released while finding the correct parent, the procedure needs to confirm that L is still underflowed before performing the merge.

Qr.Condense(L)

1. Acquire $\rho(L)$.
2. **if** $L = NULL$ **then** return.
3. Let L has n entries as E_1, \dots, E_n .
4. $SValue := V_n$ where $E_n = (L_n, V_n)$.
5. $Parent := GetParent(L, SValue)$.
6. **if** ($Parent = NULL$) and (L has no child) **then**
 - Acquire $\epsilon(L)$.

- Mark L as deleted.
 - Set root of the tree to $NULL$.
 - Release $\epsilon(L)$.
 - Return.
7. Acquire $\epsilon(Parent)$.
8. Let $Parent$ has k children as P_1, \dots, P_k including L .
9. Acquire $\epsilon(P_i)$ for $i = 1, \dots, k$.
10. if $UNDERF(L)$ then
- Let L has n children as P_{l1}, \dots, P_{ln} .
 - Acquire $\epsilon(P_{lj})$ for $j = 1, \dots, n$.
 - Merge the entries of P with its siblings.
 - Update the link and parent pointers of P_i 's.
 - Update the parent pointers of P_{lj} 's.
 - Update $Parent$.
 - if L has no child then mark L as deleted.
 - Release $\epsilon(P_i)$ for $i = 1, \dots, k$.
 - Release $\epsilon(Parent)$.
 - $Qr.Condense(Parent)$.
- else
- Release $\epsilon(P_i)$ for $i = 1, \dots, k$.
 - Release $\epsilon(Parent)$. □

5.6.4 The Secondary Trees

In the previous discussion of the quad-R tree, a bucket is simply a linear list to store rectangles which are embedded in a terminal quadrant. When there are many objects stored in a bucket, the bucket overflows and a better organization is needed. In this section, we show how to improve the quad-R tree by associating it with secondary data structures.

In the extended quad-R tree, each entry in a leaf node is associated with a secondary tree. A secondary tree is a B^+ -tree. Each non-leaf node of the tree contains entries of the

	ρ	ω_X	ϵ
ρ	1	0	0
ω_X	0	1	0
ϵ	0	0	0

Figure 5.18: Lock compatibility matrix for a quad-R tree.

form $(Value, P)$ where P is a pointer referencing a child node and $Value$ is a scalar value which is the largest scalar values amongst the entries in P . A leaf node contains entries of the form $(Value, O)$ where O is the pointer to a spatial object and $Value$ is the scalar value after transforming the MBR of O with the method described later. In later discussions, the *primary tree* refers to the original quad-R tree.

In a quad-R tree, a search operation maintains locks on the leaf nodes until the operation completely finishes. When secondary trees are used, each leaf node in the primary tree will have its entries referencing different secondary trees. Hence, the roots of the secondary trees referenced by the entries will need to be locked as well. Similarly, when an update operation inserts/deletes an object in a leaf node, the root of the corresponding secondary tree is locked. We observe that once an update operation has done its work at a leaf node of the secondary tree, the lock on the root can be released without affecting other operations. To extend the algorithms for the quad-R tree, we introduce a new lock type, the ω_X . It is the *intended-update* lock which is only compatible with itself. The new lock compatibility matrix is shown in Figure 5.18.

In general, there is no good way of ordering the MBRs of spatial objects. If the MBRs are clustered in different regions in the data space as shown in Figure 5.19, then an index structure such as R-tree can be used. However, R-tree does not work well for the situation shown in Figure 5.20. In the quad-R tree, we encode a rectangle to a higher dimensional point and store it in a secondary tree.

To construct a secondary tree, we first transform all intervals of the MBRs of data objects along the x-axis to points in a 2-d space where the 2 axes represent the left and right end points of the intervals. After the transformation, an interval I ($[x_1, x_2]$) becomes a 2-d point (l_x, r_x) where l_x is the left end and r_x is the right end of I , respectively. Furthermore, because all intervals would have their left end value smaller or equal to the right end value, the transformed 2-d points will only be in the area above the diagonal line showed in Figure

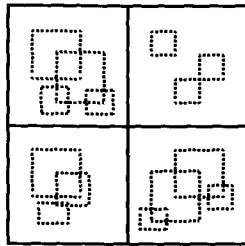


Figure 5.19: Clusters of rectangles which separated nicely.

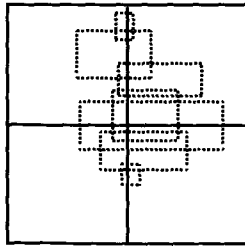


Figure 5.20: Rectangles that are hard to organize.

5.21. To determine if a given interval $[s_l, s_r]$ overlaps with I , one can perform a query in the new 2-d space with the following test:

$$(l_x \leq s_r) \text{ and } (r_x \geq s_l)$$

The query above defines a search area for all the intervals overlap with $[s_l, s_r]$ in the new 2-d space. Along the x-axis, there can be three types of intervals, A , B and C as shown in Figure 5.22. These three types of intervals correspond to the three different regions shown in Figure 5.23.

We want to linearize the points so that a window query can be mapped into a range query. Let $[x_1, x_2]$ be the x-interval of a MBR which is inside a terminal quadrant. The center of the quadrant is at $[x_c, y_c]$. The linearization can be done as follows:

If $(x_1 \leq x_c)$ and $(x_2 \leq x_c)$

$$\theta_x(x_1, x_2) = x_1 + (x_2^2 + x_2)/2 + 1$$

If $(x_1 \leq x_c)$ and $(x_2 > x_c)$

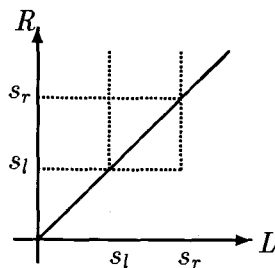
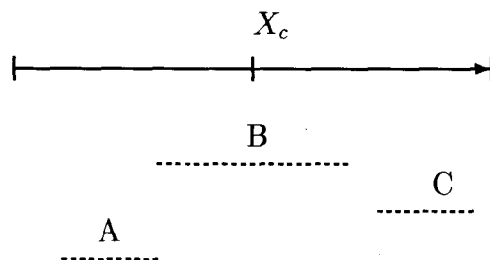
Figure 5.21: Query region for search interval $[s_l, s_r]$.

Figure 5.22: Three types of line intervals.

$$\theta_x(x_1, x_2) = x_1 + (x_2 - x_c - 1) * x_c + (x_c^2 + 3 * x_c + 2)/2 + 1$$

If $(x_1 > x_c)$ and $(x_2 > x_c)$

$$\theta_x(x_1, x_2) = (x_2 - x_1 + 1) + ((max_l + 1) * max_l - (2 * max_l * x_1 - x_1^2 + x_1))/2$$

where max_l and max_r are the largest possible number of the left and right end points of all x-intervals (see Figure 5.23). To find out the x-intervals overlaps with a given search interval $[s_l, s_r]$, we can use a range query as $[\theta_x(0, s_l), \theta_x(s_r, max_r)]$. For overlappings between y-intervals of the MBRs, similar transformations and queries can be applied. The transformation (θ_y) is defined as θ_x except the end values (bottom instead of left, top instead of right) and the center point (y_c instead of x_c) are different.

After transforming the x- and y-intervals of the MBRs, each MBR has a pair of values for its corresponding intervals in the data space \mathcal{D} . The MBR of an i^{th} object is represented as (x^i, y^i) (i.e., $(\theta_x(x_{i1}, x_{i2}), \theta_y(y_{i1}, y_{i2}))$) in the data space \mathcal{D} . To complete the encoding, we apply the bit-interleaving together with Gray code mapping to the pair in order to obtain a

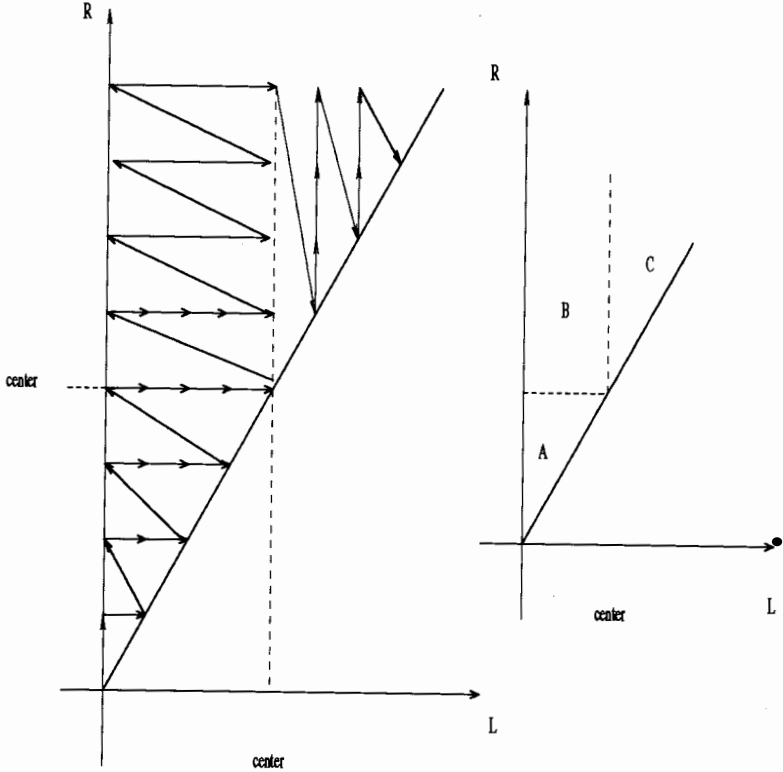


Figure 5.23: Linearization of the \mathcal{D} space.

scalar value. To simplify later discussions, we denote the encoding scheme by $\psi_q(R)$ where R is a rectangle.

Search

When secondary trees are used, a search operation continues to work in secondary trees after arriving at the leaf nodes of the primary tree. Locks on the leaf nodes in the primary tree are maintained as before. The locks are released after the search operation finishes its work in the secondary trees.

To perform a search operation on a secondary tree, we use θ_x and θ_y to obtain a query region W_R in \mathcal{D} from a given search window W . Let $Low = \min(\psi_G(W_R)) = \min\{\psi_G(x) \mid x \text{ is a point in } W_R\}$ and $High = \max(\psi_G(W_R)) = \max\{\psi_G(x) \mid x \text{ is a point in } W_R\}$. We can then represent W_R by the range $[Low, High]$. The range can be decomposed into sub-ranges of values such that the values outside the sub-ranges represent regions that do not intersect with W_R . Suppose the associated scalar value of a spatial object is V . If V falls into $[Low, High]$ at a leaf node, then the object is tested whether it overlaps with the search window. The computations of $High$ and Low are similar to those for the B^+ -tree in Chapter 4.

More formally, the search operation on a secondary tree is invoked by the procedure $Qr.SearchS(W, R, Low, High, OldValue, LeafStack)$, where W is the search window, R is a node of the tree, Low is the starting end of the query range, $High$ is the higher value of the range, $OldValue$ is the largest number in the node visited previously, and $LeafStack$ is the stack contains the references of leaf nodes. Initially, $OldValue$ is the value of the largest possible number, and $LeafStack$ is empty.

$Qr.SearchS(W, R, Low, High, OldValue, LeafStack)$

1. Acquire $\rho(R)$.
2. **if** R is a non-leaf node **then**
 - Let $\{E_i = (R_i, V_i) \mid i = 1, \dots, k\}$ be the entries in R such that $V_{i-1} < Low \leq V_i$.
 - **if** $(Low > V_n)$ and $(NEXT(R) \neq NULL)$ **then**
 - $R' := NEXT(R)$.
 - $Qr.SearchS(W, R', Low, High, OldValue, LeafStack)$.


```

    else Qr.SearchS(W, Ri, Low, High, Vi, LeafStack).
    • Release  $\rho(R)$ .

else

    • Return all objects in R inside W.
    • PUSH(R, LeafStack).
    • R := NEXT(R).
    • Acquire  $\rho(R)$ .
    • Let R has n entries.
    • while (R ≠ NULL) and (V1 ≤ High) do
        – Return all objects in R inside W.
        – PUSH(R, LeafStack).
        – R := NEXT(R).
        – Acquire  $\rho(R)$ .
    enddo
    • if (R ≠ NULL) then release  $\rho(R)$ .

```

□

Insert

When an insert operation reaches a leaf node, say *P*, at the primary tree, it acquires an ω_X -lock on the node. This would disallow a search operation to work on *P* but other update operations can continue to work in the corresponding secondary trees simultaneously. After the object has been inserted at a leaf node in the secondary tree, the ω_X -lock is released.

Similarly to the primary tree, there are two phases in a secondary tree for an insert operation. The implementations are the same as in the primary tree. Therefore, we only show the main algorithm, *Qr.InsertS*, for the insert operation here. The first phase of work is carried out by *Qr.SelectS* and the second phase is done by *Qr.CleanUpS*. More formally, *Qr.InsertS*(*O*, *R*) is carried out as follows, where *O* is the object to be inserted, *R* is the root of the secondary tree initially and a new entry E_O will be added to the selected leaf node.

Qr.InsertS(*O*, *R*)

1. *Low* := $\psi_q(\text{MBR}(O))$.

2. $L := \text{Qr.SelectS}(O, \text{Low}, R)$.
3. Upgrade $\rho(L)$ to $\epsilon(L)$.
4. Let E_O be the new entry for O .
5. Add entry E_O in node L .
6. Downgrade $\epsilon(L)$ to $\rho(L)$.
7. $\text{Qr.CleanUpS}(L, \text{Low})$.

□

Delete

A delete operation works similarly to an insert operation in the previous subsection. When a delete operation reaches a leaf node at the primary tree, it acquires an ω_X -lock on the node to avoid any search operation active on the node. After an object is removed from a leaf node in the secondary tree, the ω_X -lock is released.

In a secondary tree, the first phase of a delete operation is to locate the leaf node L which contains the object O and deletes it afterwards. It is done by the procedure Qr.FindS . In the second phase, we use the procedure Qr.CondenseS to perform the required updates, if necessary. The “mark-and-remove” approach is used to delay the actual removal of the nodes. Similarly to the insertion algorithm above, only the main procedure, Qr.DeleteS , is presented here. There are two arguments for the Qr.DeleteS where O is the object to be deleted and R is the root of the tree initially.

$\text{Qr.DeleteS}(O, R)$

1. $\text{Low} := \psi_q(\text{MBR}(O))$.
2. $L := \text{Qr.FindS}(O, \text{Low}, R)$.
3. **if** $L \neq \text{NULL}$ **then**
 - Upgrade $\rho(L)$ to $\epsilon(L)$.
 - Let E_O be an entry referencing O .
 - Remove entry E_O in node L .
 - Release $\epsilon(L)$.
- else** release $\rho(L)$.
4. $\text{Qr.CondenseS}(L)$.

□

5.6.5 Correctness of the Operations

In this section, we would like to show the correctness of the operations in a quad-R tree. We first work on the primary tree and extend the proofs to include the secondary trees later. A slightly different approach, other than the one used in Section 4.1.5, is used to prove the correctness of the operations. We say that a protocol \mathcal{P} on an index structure is **correct** if it satisfies the following two conditions.

1. \mathcal{P} is deadlock free, and
2. In any execution \mathcal{E} that \mathcal{P} generates, each search operation $S(W)$ returns object O iff O is available to it in \mathcal{E} .

In order to prove the second condition, it is sufficient to show the following:

- (a): If an object O is available to $S(W)$, then at least one subsearch maintains a window W' such that $O \in W'$, until O is returned.
- (b): The operations form a serializable schedule.

Note that, the above conditions are the same as those in Section 4.1.5 except for (b). In the rest of this section, an operation means an insert, delete or a subrange query on a leaf node. Furthermore, each operation obtains a timestamp when it first accesses the root.

Lemma 5.10 *An operation can overtake any other operation.*

Proof: During a downward traversal of a search operation, at each step, it acquires a ρ -lock on a node (P), obtains the relevant information, releases the lock on the node and attempts to work on the next one (P_1). Throughout its work, only a single lock is held. Therefore, a second operation traversing along the same access path can lock the node P_1 before the search operation (i.e., it overtakes the search operation). Similarly, in the first phase of an update operation, only one lock is acquired at a time and other operations can overtake it. \square

Theorem 5.9 *The protocol QR for the quad-R tree is deadlock free.*

Proof: Assume that a set of operations is deadlocked under the protocol QR, and let Δ be the set of all deadlocked operations. We derive a contradiction out of this assumption. Let T be an operation in Δ .

- **a:** T is a search operation. During the downward traversal, T only acquires a single lock at each step. Hence, it cannot be blocked. When a search operation reaches the leaf level, it may lock multiple nodes, say P_1, \dots, P_k . The nodes are ρ -locked from left-to-right at the leaf level. Suppose there is an update operation, U , works at leaf node P_i . If it ϵ -locks P_i first, T needs to wait. U may cause P_i to be full or deleted, but it will release the ϵ -lock on P_i before any tree reorganization. During the reorganization, the locks are acquired from top-down and left-to-right (i.e., in order of P, P_1, \dots, P_k) which is in the same order as for T at the leaf level. Hence, partial locking cannot occur. If T acquires the lock on P_i first, T can continue. Therefore, in either case, deadlock cannot occur.
- **b:** T is an insert operation in its first phase. Since T only acquires a single lock at each step, it cannot be blocked forever.
- **c:** T is an insert operation in its second phase, executing *QR.CleanUp*. In this case, T is trying to ϵ -lock node P , P 's child and grandchild nodes. Before T has successfully acquired the first ϵ -lock, it holds no lock at all. It acquires the locks in the top-down and left-to-right order. If there is another update operation (U_1) trying to work on P , the order of execution depends on the order of acquiring the lock on P . There cannot be any deadlock between T and U_1 because of the lock acquisition order. A different operation may have placed locks on child nodes and/or on grandchild nodes of P , but it will not block T because of the order of lock acquisition as well. Thus, T cannot be blocked forever. Hence, a contradiction to the assumption.
- **d:** T is a delete operation in its first phase. The argument is the same as in (b).
- **e:** T is a delete operation in its second phase. The argument is the same as in (c). \square

Lemma 5.11 *An update operation will correctly modify the quad-R tree.*

Proof: An update operation correctly modifies the tree if the tree structure is preserved after the operation and it does not cause any later search operation to visit the wrong nodes of the tree.

During the first phase of an update operation, the tree is not modified at all. When there is only one update operation, the modification is trivially correct. Therefore, we will

only need to consider the situation where there are multiple update operations backtracking to perform tree reorganization.

Let two update operations, U_1 and U_2 trying to work on P as shown in Figure 5.24. Due to the order of lock acquisition, U_2 (U_1) will need to wait if U_1 (U_2) ϵ -locks P first. Hence, U_1 and U_2 cannot modify the tree at node P at the same time. Another case is where U_1 works on P and P_1 while U_2 works on P_1 . If U_2 ϵ -locks P_1 first, U_1 will wait. When U_2 finishes, P_1 will either be added or deleted an entry. This does not affect the work of U_1 because it will check if P_1 is still underflowed or overflowed before continuing its work (see procedure *Qr.CleanUp* and procedure *Qr.Condense*). If U_1 ϵ -locks P_1 first, U_2 will need to wait. If P_1 is split into P_1 and P_2 , U_2 can use the link pointer at P_1 to find the correct node, if necessary. If P_1 is removed, U_2 can find the correct parent by using the head link pointer at the level of P_1 (see procedure *GetParent*). Therefore, in the cases above, both U_1 and U_2 can proceed to modify the tree correctly. \square

Lemma 5.12 *In the protocol QR, no search operation will miss an available object.*

Proof: We prove the lemma by showing that at every level, a search operation will find out the correct node to visit at the next lower level in the tree. Furthermore, in a quad-R tree, it is always true that the values in the nodes at the same level follow an ascending order from left-to-right.

In Figure 5.24, a search operation S_1 visits P . It will first acquire a ρ -lock on P and decides which node to visit next. Let the node be P_1 and its entry in P is (V_1, P_1) . Suppose, there is an update operation U_1 which is about to modify P_1 . If S_1 ρ -locks P_1 first, P_1 will not be modified and S_1 can continue as intended. P_1 can only be modified by U_1 when U_1 has ϵ -locked P_1 . If this is true, then after U_1 has modified and released the lock, the new largest value in P_1 will be V'_1 ($\leq V_1$). If $V'_1 = V_1$, then S arrives at P_1 as intended. If $V'_1 < V_1$, then S can use the link pointer to find the correct node whose largest value is greater than or equal to V_n . This is possible because a link pointer is introduced simultaneously at the time of splitting a node.

Suppose U_1 is to delete P_1 . If S_1 ρ -locks P_1 before U_1 , S_1 will visit the node as intended. If U_1 acquires the lock on P_1 first and removes it, S_1 will find P_1 as marked and it will use the head link pointer at the current level to start finding the correct node via the link pointers of nodes at the current level (see procedure *FindChild*).

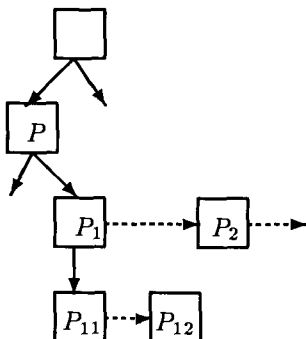


Figure 5.24: Restructuring of a subtree in a quad-R tree.

Hence, a search operation is always able to find the correct node to visit at each level of the tree. Consequently, the operation will reach the leaf nodes referencing the available objects. \square

Theorem 5.10 *The protocol QR for the quad-R tree is correct.*

Proof: From Theorem 5.9, the protocol QR is deadlock free. From Lemma 5.12 and Lemma 5.11, all operations traverse the tree correctly. Hence, we now show that the operations form a serialized schedule.

Let \mathcal{T} be the completion order of the operations. If \mathcal{T} has no update operation, then \mathcal{T} is trivially serializable. If \mathcal{T} has no search operation, then \mathcal{T} is also serializable according to the sequence of the times when the update operations accessed the leaf nodes. Note that individual subsearch operations belonging to a search operation can proceed independently, since they are executed in parallel. Thus, a search operation (S) can have several completion times in \mathcal{T} .

Let S_1, \dots, S_n be the search operations and U_1, \dots, U_m be update operations in \mathcal{T} . If, in \mathcal{T} , all search operations are completed consecutively and there is no update operation in between two search operations, then \mathcal{T} will be one of 3 possible schedules below: $[S_1] \dots [S_n][U_1] \dots [U_m]$, $[U_1] \dots [U_m][S_1] \dots [S_n]$ and $[U_1] \dots [U_i][S_1] \dots [S_n][U_{i+1}] \dots [U_m]$. Each of the schedules has two or three parts. As there is only one type of operations, it can be serialized as discussed in the previous paragraph. Hence, \mathcal{T} is serializable. Therefore, the interesting situation is where the completion order of the operation is mixed, such as $[U_1][S_1][S_2][U_2] \dots [S_n][U_{m-1}][U_m]$.

Assume that \mathcal{T} is not serializable under the protocol QR. This implies that there are at least two operations, S_1 (a search operation) and U_1 (an update operation), have an order as T_1 ($[S_1][U_1][S_1]$) or T_2 ($[U_1][S_1][U_1]$). However, T_2 cannot occur because each update operation can only work on a single leaf node. In the case of T_1 , if U_1 and S_1 visit no common leaf node, we can re-arrange T_1 to $[U_1][S_1]$. Suppose U_1 and S_1 visit some common leaf nodes. In our implementation, S_1 does not release the locks on the leaf nodes until the end. U_1 can only modify a leaf node before or after S_1 has completely finished. Hence, a contradiction to the assumption. Therefore, \mathcal{T} is serializable.

From the discussion above, we conclude that the protocol QR is correct. \square

Theorem 5.11 *The protocol QR extended with secondary trees is correct.*

Proof: The algorithms for the operations in the secondary trees are similar to those in the primary tree. The only difference is that an update operation (U_1) needs to hold its lock on a leaf node (P) of a primary tree. The ω_X -lock is used by U_1 to disallow a search operation (S) to work on P node and forces S to wait. After U_1 has finished its first phase in the corresponding secondary tree, its ω_X -lock, which is placed on the root of the secondary tree, is released. As there is no change to the leaf node in the primary tree, updating in the primary tree is not necessary.

In the extended tree, update operations are using the same protocol except one extra lock type (the ω_X). As ω_X -locks are compatible to itself, it does not create any deadlock between update operations. When an ω_X -lock is placed on a leaf node of the primary tree, it has a similar effect to a search operation as an ϵ -lock. Therefore, we can use the same proof presented in Theorem 5.9 and there is no deadlock created.

Algorithms developed for the primary tree are used for the secondary trees. All operations on a secondary tree behave the same as those on the primary tree. Therefore, we can extend the proofs used for the primary tree to includes the secondary trees. \square

Chapter 6

Comparisons and Implementations

In this chapter, we compare the different index structures which we have presented in previous chapters. In Chapter 4, we have discussed four index structures for point data adopting two different concurrency control approaches. In Chapter 5, we have discussed two index structures for rectangular data adopting 5 different concurrency control approaches. Amongst the index structures, the B⁺-tree uses no spatial relationships between data objects. The R-tree allows bounding rectangles of nodes at the same level of the tree to overlap. On the other hand, in a K-D-B tree, the nodes at the same level of the tree cover non-overlapping regions. We have also studied some hybrid structures, such as the quad-B tree, which is a variant of a quadtree but allows overlapping of regions of nodes at the same level. Besides the spatial relationships, the selection of the concurrency control approaches to be used on an index structure affects the search performance of the index structure. During our work, we realize that there is no particular index structure that performs the best in all circumstances. The selection of an index structure and the concurrency control approach depends on the distribution of the spatial data and the types of operations. In Figure 6.1, we summarize the characteristics of the index structures presented in the previous chapters.

Even though we have developed different index structures for point and rectangular data, their principle characteristics and problems are similar. To compare them, we look into the following two aspects:

- Locking protocols.
- Storage utilization.

For the rest of the chapter, we will investigate the behavior of the index structures according

	Data Type	Concurrency control approach	Overlapping	Object order
B ⁺ -tree	point	lock-coupling	-	Yes
R-tree	point	lock-coupling	Overlap	No
K-D-B tree	point	lock-coupling	No-overlap	No
Quad-B tree (I)	point	lock-coupling	Hybrid	Yes
Quad-B tree (II)	point	link pointers	Hybrid	Yes
R-simple tree	rectangle	single lock	Overlap	No
R-lock tree	rectangle	modify lock	Overlap	No
R-couple tree	rectangle	lock-coupling	Overlap	No
R-opt tree	rectangle	give-up	Overlap	No
R-link tree	rectangle	link pointers	Overlap	No
Quad-R tree	rectangle	link pointers	-	Yes

Figure 6.1: Index structures studied in Chapter 4 and 5.

to the aspects above. At the end, we present the implementation results of some of the index structures.¹

6.1 Locking Protocols

In total, there are 11 different combinations of the index structures and concurrency control approaches. To compare them, we look into the overhead due to lock types, the number of locks acquired, the scope of an operation and the overtaking of operations.

The overhead for lock management is less when there are a smaller number of lock types. Consequently, the performance of operations improves. With the lock-simple approach, there are only two types of locks: a share (ρ) lock to support multiple search operations and an exclusive (ϵ) lock to allow a single update operation at a time. The same types of locks are used again for the lock-modify approach in the R-lock tree and for the give-up approach in the R-opt tree. With the lock-coupling approach, a third lock type, the warning lock (ω), is introduced to improve the level of concurrency by allowing search operations to enter the scopes of update operations. The link approach does not need the warning locks as operations can overtake each other. However, in the quad-B tree where a hybrid approach is used, link pointers are used by search operations while lock-coupling is adopted by update

¹The SR program is available by sending a message to either tyvng@cs.sfu.ca or tyvng@terryfox.ubc.ca.

operations. In quad-B tree, two additional types of locks (ω_d and χ) are introduced. They are mainly used to avoid overtaking between search and delete operations, which may cause non-serializability (see Section 4.1.5). In the quad-R tree, search operations perform range queries rather than window queries as in other index structures. With this new approach, overtakings between operations are allowed. Hence the new lock types used in the quad-B tree are not necessary. Besides the ρ -lock and ϵ -lock, in order to improve the support of multiple update operations, we proposed a new lock type (the ω_X) for the secondary trees associated with the quad-R tree.

Throughout our study, we have assumed that search operations are the most frequent operations. When there are few or no update operations, we want the search algorithms to have minimal overhead. This has led us to use a small number of locks at each step of a search operation. The lock-coupling approach may not be an efficient approach because at each step of a search operation it will hold locks on a node and all its child nodes at the same time. Even worse, in a quad-R tree, a given query range may span all possible values, which causes a search operation to acquire locks on all the leaf nodes. With the use of link pointers in the quad-B tree and the range information in the R-opt tree, the trees only need to acquire at most two locks at each step of a search operation. If there are only search operations active in the tree, the R-simple tree, which uses a single lock for each operation, is the best.

In most database systems, some updating is required from time to time; otherwise, concurrency control algorithms are not needed. Even if update operations are infrequent, we do not want to lock too many nodes when an update operation is active. During the first phase of an update operation, its lock acquisition behavior is similar to a search operation. The index structures we studied can be divided into four different groups according to the number of nodes locked during the second phase of an update operation. The best group consists of the index structures which adopt the lock coupling approach. At each step during reorganization, an update operation will lock at most one node and all its child nodes (i.e., a total of $1 + M$ nodes). However, in the R-tree variants, frequently there is no single deletion path for a delete operation. Hence, the parent pointers are used and this makes every step of reorganization involve a node, its child nodes and its grandchild nodes (i.e., a total of $1 + M + M^2$ nodes). In the R-link tree, because of the use of link pointers and because there is no ordering amongst nodes in the tree, a reorganization step can involve a sub-tree of more than 3 levels ($> 1 + M + M^2$ nodes). In a K-D-B tree, many nodes ($> 1 + M + M^2$) may

	No. of lock types	Max. no. of nodes locked by a search oper.	Max. no. of nodes locked by an update oper.
B ⁺ -tree	3	$1 + M$	$1 + M$
R-tree	3	$1 + M$	$1 + M + M^2$
K-D-B tree	3	$1 + M$	$> 1 + M + M^2$
Quad-B tree (I)	3	$1 + M$	$1 + M$
Quad-B tree (II)	5	2	$1 + M + M^2$
R-simple tree	2	1	<i>The whole tree</i>
R-lock tree	2	1	<i>The whole tree</i>
R-couple tree	3	$1 + M$	$1 + M + M^2$
R-opt tree	2	2	$1 + M + M^2$
R-link tree	2	1	$> 1 + M + M^2$
Quad-R tree	3	$1 + n$	$1 + M + M^2$

Figure 6.2: Summary characteristics of locking protocols in the index structures (M is the maximum number of entries within a node and n is the number of all the leaf nodes).

need to be locked because of forced splitting. Among all the index structures, the R-simple tree and the R-lock tree are the worst, since the whole tree is locked during reorganization. In Figure 6.2, we summarize the above discussion.

In the previous paragraphs, we have discussed the number of locks needed at each step of an operation. At first glance, these numbers appear to serve as good indicators of overhead in the algorithms. However, when there is a mix of search and update operations, the fact that search operations can be active in the scope of an update operation becomes an important factor. If a search operation cannot work within the scope of an update operation, it is blocked and has to wait until the update operation finishes. Amongst all the index structures, the R-simple tree is the worst. Index structures adopting lock-coupling allow a search operation to enter the scope of an update operation but overtaking is not permitted. For the quad-B tree (II), the R-link tree, and the quad-R tree, which use the link approach, a search operation can both overtake and enter the scope of an update operation. In Figure 6.3, we summarize the characteristics of the locking aspect of the index structures discussed so far.

The link technique has been reported in [SrC91, JoS93] as having a good performance due to the small numbers of locked nodes at a time in B⁺-trees. However, when we developed

	Overtaking between search and update	Search in the scope of update	Update in the scope of update
B ⁺ -tree	No	Yes	No
R-tree	No	Yes	No
K-D-B tree	No	Yes	No
Quad-B tree (I)	No	Yes	No
Quad-B tree (II)	Yes	Yes	No
R-simple tree	Yes	No	No
R-lock tree	Yes	Yes	No
R-couple tree	No	Yes	No
R-opt tree	Yes	Yes	No
R-link tree	Yes	Yes	Yes
Quad-R tree	Yes	Yes	Yes

Figure 6.3: Summary characteristics of concurrent operations which can be active in the index structures.

the *R-link* tree in [Ng94b], we realized the difficulties of re-arranging link pointers during insertion and deletion. It is difficult because there is no specific order amongst nodes at the same level of a tree. This is true both for rectangular data and point data. Rearrangement of link pointers propagates down the tree. In the R-link, during tree reorganization, a subtree is often needed to be exclusively locked. If deletions are rare, the R-link tree can be a good candidate as a spatial index because of its efficient search and insert algorithms.

A further consideration is the overlapping property of a spatial index. In the R-trees, the MBRs of the nodes can overlap and data space is not divided into non-overlapping partitions. When a new spatial object is added to an R-tree, besides the changes to the MBRs in the tree, there is little complexity in organizing the entries in a node of the tree. In the K-D-B tree described in Chapter 4, when a new spatial object causes an overflow, the re-partitioning of the data space is often required. If it happens, each reorganization step will work on many nodes besides the current node and its two levels of descendants. Furthermore, the up-and-down splitting of nodes in the tree can occur as well. We have solved the recursive splitting by using forced splits, but it then causes poor storage utilization.

6.2 Storage Utilization

In the previous section, we discussed the consequences in performance when different locking protocols are used for different index structures. As usual, there is always a trade-off in the design of an index structure: performance versus storage. In the following discussion, m (M) is the minimal (maximal) number of entries in a node of a tree.

The B⁺-tree and most of the R-tree variants we discussed are always balanced and the number of entries in each node is in the range $[m, M]$. The quad-R tree is the index structure which is not balanced because of the associations of secondary trees at the leaf nodes of the primary structure. In both the primary tree and secondary trees, each node, except the leaf nodes in the primary tree and the roots, may have the number of entries in the range $[m, M]$. The nodes in a quad-B tree behave similarly and only the leaf nodes and the root will have less than m entries. The R-lock tree is another index structure whose leaf nodes can have their numbers of entries outside $[m, M]$. This is due to the delay in tree reorganization. A leaf node may have overflowed or underflowed because of several insert or delete operations visiting the node previously. An improvement can be made by adjusting the size of the maintenance queue but it will cause the whole tree to be locked more frequently for reorganization. In a K-D-B tree, its non-partitioning property causes splits to cascade over lower levels and may create lower level nodes which are usually empty. There is no guarantee that each node has at least m entries. Therefore, we expect it to have the worst storage utilization.

Each concurrency control approach requires some additional data structures at the nodes of an index structure. The R-simple tree has no overhead but a lockable variable. In the R-lock tree, the overhead is two counters and a maintenance queue. As for the other R-trees, because of the backtracking due to update operations, each node needs to maintain the parent pointer. In the quad-B tree (II), which adopts both link technique and lock-coupling, a link pointer is added to each node as well. Similarly, the quad-R tree uses a link pointer and a parent pointer at each node. So far, these index structures have relatively small storage overhead when compared with the R-opt tree and R-link tree. In an R-opt tree, each node has a version number and a change type while each entry has the version number of the node it references. Furthermore, every search operation maintains a log table, which contains references to nodes in the R-opt tree. The table is used to store the range information which is used to facilitate the validation of the operation. Similarly to R-opt

	Tree balance	No. of entries between $[m, M]$	Extra data Structure at each node
B ⁺ -tree	Yes	Yes	No
R-tree	Yes	Yes	Yes
K-D-B tree	Yes	No	No
Quad-B tree (I)	Yes	No	No
Quad-B tree (II)	Yes	No	Yes
R-simple tree	Yes	Yes	No
R-lock tree	Yes	No	No
R-couple tree	Yes	Yes	Yes
R-opt tree	Yes	Yes	Yes
R-link tree	Yes	Yes	Yes
Quad-R tree	No	No	Yes

Figure 6.4: Summary characteristics of storage utilization in the index structures (M is the maximum number and m is the minimum of entries within a node).

tree, in an R-link tree, besides pending updates and link pointers, a log table is used to record all operations in order to ensure serializability discussed in Section 4.1.5.

6.3 Implementations

To assess the performance of the concurrency control methods described in the previous chapters, we have implemented some of the algorithms in **SR**[And82]. SR is a language for writing distributed programs. The main language constructs of SR are **resources** and **operations**. The resource is similar to the **class** in an object-oriented language like C++, which specifies available procedures, processes and shared data. Creating an instance of a resource is analogous to creating an object belonging to a class. SR also supports dynamic process creation, message passing, multicast, and semaphores.

In our implementations, we use the client-server paradigm. The server is implemented as a set of multi-threaded processes to serve *insert*, *search*, and *delete* requests on an R-tree. A new process is created to provide a service whenever a request is received. Of the several resources in our implementation, the lock manager is implemented in the resource **Lock** which handles all lock requests. The scheduling of the processes is handled by the runtime library of the SR language.

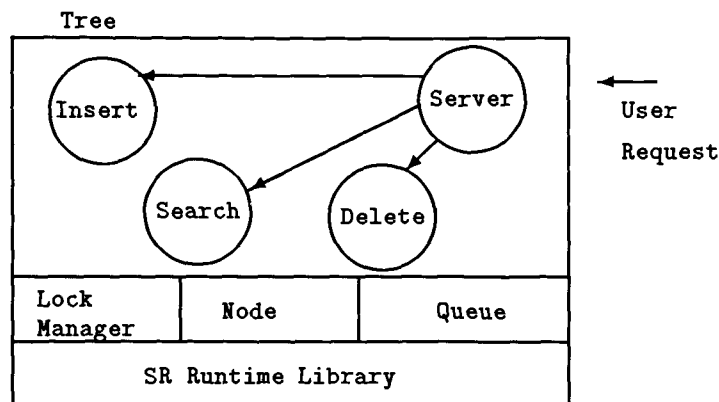


Figure 6.5: Implementation Structure

We tested our simulation programs on a SUN SPARC 10 Workstation with 64 Mbytes of main memory for rectangular data and point data. In our simulations, we generated only search and insert requests. The reason for this decision is because we thought deletions and insertions required much the same service. We were mainly interested in measuring the search performance under different locking methods, since, presumably, they comprise the majority of operations in most applications. Batches of pre-generated requests, composed of varying numbers of insertions and searches, were tested against trees of different sizes. For each batch, we calculated the average response time (*atime*) of all the search operations.

6.3.1 Point Data

We have implemented the search and insert algorithms for the B^+ -tree, R-tree, and K-D-B tree. There are four basic resources for the programs. The first basic resource we use is **Geometric** which defines the data structures of a rectangle and a point and their associated operations. The second resource, **Node**, specifies the content of a node in a tree and the related node operations. For the three different trees, this resource uses generic procedures and similar data structures. This may not be efficient for a particular type of index structure, but it provides fair comparisons for later simulation. **Lock** is the third basic resource which represents a lock manager. The last resource, **Queue**, specifies linear lists and stacks. Based on the four resources, a **Tree** resource is tailor-made for each type of trees described before (see Figure 6.5).

In each simulation run, initially, a tree was created with a data file containing only

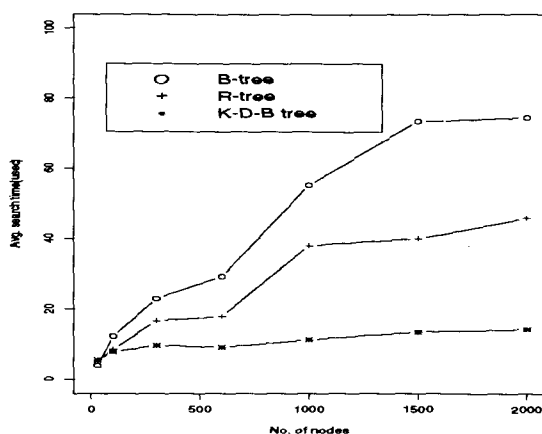


Figure 6.6: Search performance with no insert operation. (Average search time in μ seconds versus the number of nodes).

insert requests. The locations of the inserted points were generated randomly within a 2-dimensional space which measured 1000×1000 . The search windows of the search operations were generated using a uniform distribution over the range $[1, 10]$. The maximum number of entries in each node, M , was set to 5 (a relatively small number), so that there would be sufficient number of levels in the tree with a relatively small number of nodes that the memory capacity allowed.

For each type of tree, we first generated 7 different initial trees (of sizes 30, 100, 300, 600, 1000, 1500, and 2000), and then ran simulations with two different request batches, yielding 14 values of *atime*. The first batch consisted of 100 search operations with no insert operation. Each operation in the batch was submitted sequentially. As shown in Figure 6.6, the K-D-B tree performed best and the R-tree performed better than the B^+ -tree. The second batch we tested consisted of 50 insert operations randomly interleaved with 50 search operations. The results shown in Figure 6.7 indicate that, among the three index structures, the K-D-B tree again performed best.

Reviewing the above results, we believe that it is the non-overlapping property of the K-D-B which contributes to the speed of its search performance. In the K-D-B tree, a search operation may give rise to several sub-searches, but they search in separate regions of the data space. On the other hand, for the B^+ -tree and R-tree, we can have several sub-searches working in the same region. In the B^+ -tree, this may be due to the extra

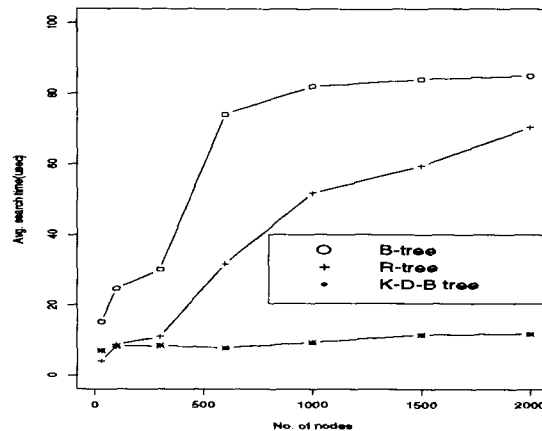


Figure 6.7: Search performance with 50 insert operations. (Average search time in μ seconds versus the number of nodes).

space resulting from the transformation of a given search window into a one-dimensional range. When the range covers extra space, it requires unnecessary sub-search operations. For example, if a search window W is represented by the diagonally opposite corner points (01,01) and (11,11), W covers 4 points. However, the range will have a length of 12 points which is 3 times the area covered by W (see Figure 4.3 in Section 4.1.1). We feel that we can probably improve the performance of the B^+ -tree by first decomposing the search window W into subwindows such that each subwindow maps to a contiguous range. We plan to investigate this possibility in future. In the R-tree, although the MBRs of its nodes overlap, a search operation may quit at an internal node once it realizes the search window no longer intersects with the node's MBR. This can be the reason why R-tree has better search performance than the B^+ -tree.

In order to have a better understanding of the results, we have measured the storage utilization of the three index structures. Figure 6.8 shows the average occupancy rate in percentage (relative to M) of the trees. For the K-D-B trees, the utilization is mostly around 50% and below 60%, while the other two index structures have better utilization. These results show the trade-off between the search performance and the required storage for the different index structures. In the implementation of the K-D-B tree, we have avoided the up-and-down updating by forced splittings. This may result in many under-utilized nodes and low storage utilization but it cuts down the work for the updating phases of insert

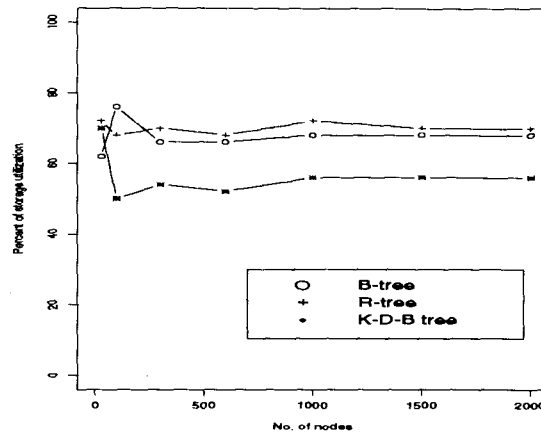


Figure 6.8: Storage utilization of different index structures. (Average node size versus the number of nodes).

operations. The B^+ -tree and R-tree have better storage utilization because they have no under-utilized nodes (except the roots). However, the K-D-B tree has much better search performance than the other two structures. Since the cost of memory and secondary storage is coming down quickly, the K-D-B tree can be a good candidate for window search queries.

6.3.2 Rectangular Data

We have implemented the search and insert algorithms for the R-simple tree, R-lock tree and R-couple tree. Similar to the implementation for the point data, we have written programs for the three locking methods using four basic resources. The first basic resource we use is the **Rectangle** which defines the data structure of a rectangle and its associated operations. The second resource, **Rnode**, specifies the content of a node in the R-tree and the related node operations. **Lock** is the third basic resource which represents a lock manager. The last resource, **Queue**, specifies linear lists and stacks. Based on the four resources, we implemented different **Rtree** resources for each locking method (see Figure 6.9).

In each simulation run, initially, an R-tree was created with a data file containing only insert requests. The locations and sizes of the inserted rectangles were generated randomly within a 2-dimensional space which measured 1000×1000 . The width and the height of each rectangle were generated using a uniform distribution over the range $[1, 10]$. The maximum number of entries in each node, M , was set to 3, so that there would be sufficient

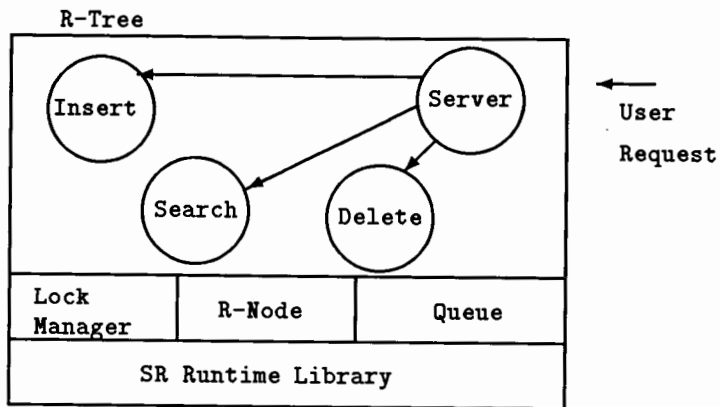


Figure 6.9: An R-tree implementation.

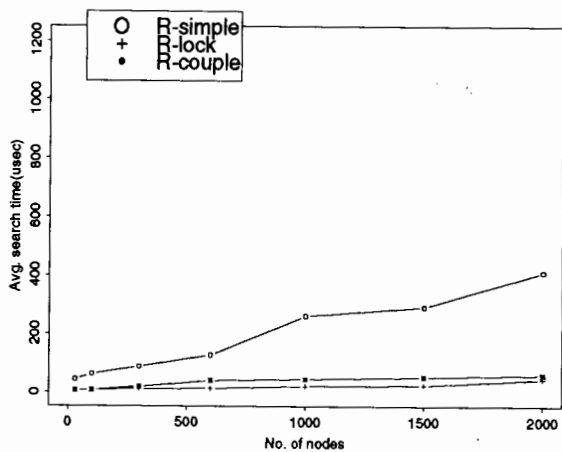


Figure 6.10: Search performance with 10% insert operations (Avg. search time in μ seconds versus no. of nodes).

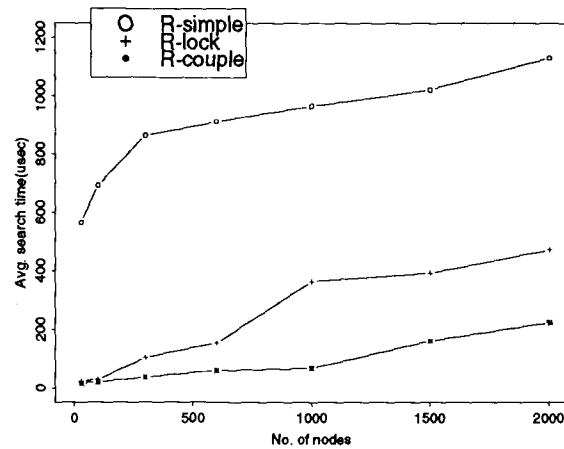


Figure 6.11: Search performance with 50% insert operations. (Avg. search time in μ seconds versus no. of nodes).

levels in the tree without too many nodes. In splitting a node, the *linear splitting strategy* of [Gut84] was used in all three programs.

For each of the 7 different initial trees (of sizes 30, 100, 300, 600, 1000, 1500, 2000), we ran simulations with two different request batches, yielding 14 values of *atime* for each locking method. The first batch consisted of 10 insert operations and 90 search operations. The insert operations are randomly interleaved with the search operations. The width and the height of the inserted object as well as the search windows of the search operations were generated using a uniform distribution over the range [1, 10]. Figure 6.10 shows that the **lock-modify** method performed best and the **lock-coupling** method performed better than the **lock-simple** approach. The second batch we tested consisted of 50 insert operations and 50 search operations. The results shown in Figure 6.11 indicate that the search times for this batch are generally higher than those in Figure 6.10. However, among the three locking approaches, the **lock-coupling** method performed best instead of the **lock-modify** method. For both experiments, the **lock-simple** method performed the worst.

The different results of the two simulations are related to the percentage of insert operations within a batch. When there are only few insertions, there will be infrequent splitting of the nodes in an R-tree. Hence, with the lock-modify method, the R-tree is not always completely locked and the overhead of the lock-coupling method becomes significant in affecting the search performance. On the other hand, when there are many insert operations, there

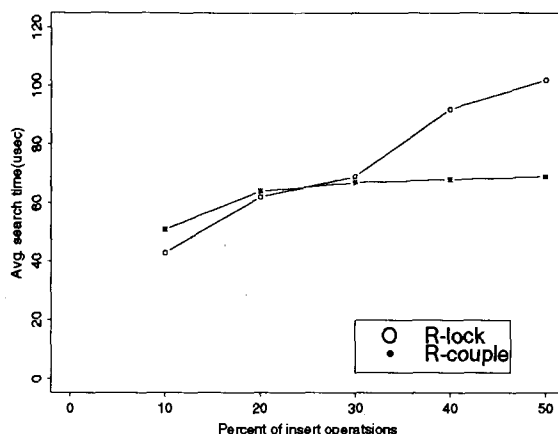


Figure 6.12: Search performance with an R-tree containing 1000 objects (Avg. search time in μ seconds versus no. of nodes).

will be many splittings and reorganizations of the R-tree. This will cause the lock-modify method to lock the whole tree frequently and a search operation will always be blocked by insert operations. In this case, the effect of the overhead of the lock-coupling method is relatively smaller.

In order to verify the above explanation, we carried out the third set of simulation experiments. Initially, we constructed an R-tree containing 1000 objects. Five batches, which consisted of 10% to 50% of insertions among 100 operations, were tested. As in the other two sets of experiments, the width and the height of the inserted object as well as the search windows of the search operations were generated using a uniform distribution over the range $[1, 10]$. With these experiments, we intended to find where the lock-coupling method has a better search performance than the lock-modify method. Figure 6.12 shows the results. The two curves crossed at approximately 25% of insert operations. Therefore, the lock-modify method appears to perform better when the percentage of insert operations is lower than 25%, and the lock-coupling method should be used when there are many (25%+) insert operations.

Chapter 7

Conclusion

7.1 Summary

This thesis extends spatial index structures to allow concurrent operations. We have studied index structures for point data and rectangular data. During our work on point data, we are interested in investigating the different spatial properties in different index structures. The lock-coupling approach is adopted to all the index structures for point data. We later changed our focus when we studied rectangular data. We used different concurrency control approaches to the same index structure (the R-tree). In addition, we developed the quad-R tree because it can be adopted to the link technique effectively. The research is limited to 2-dimensional objects, but the algorithms can be extended to higher dimensions along the same principles.

7.1.1 Point Data

We studied four different index structures for point data, adopting the lock-coupling technique to support concurrent operations. The **B⁺-tree** is used as a basis for comparison with other index structures. In this index structure, data points are transformed into scalar values and a window query is mapped into range queries. Depending on the transformation selected, the “distance” between two different point varies. We decided to use the Gray code with bit-interleaving to minimize this “distance”.

Adopting the same lock protocol, we next studied the R-tree. An R-tree tries to organize the spatial objects into clusters. However, because of multiple search paths, the search

performance is not always satisfactory. In order to improve the search performance, we then studied the K-D-B tree. In the K-D-B tree, the data space is partitioned into non-overlapping regions, which reduces the number of unnecessary search paths. However, it introduces two new problems: additional storage and the up-and-down updating propagations.

We have implemented the three index structures mentioned above using the language SR. The results indicate that the K-D-B tree has a good search performance but a poor storage utilization. We believe that this is due to the non-overlapping property of the tree. Therefore, we developed a new index structure, the **quad-B tree**, which has both overlapping (at internal nodes) and non-overlapping (at leaf nodes) properties. It is a variation of a quadtree where the terminal quadrants of the quadtree are referenced by scalar values. The quad-B tree has three distinguishing characteristics:

- The keys in the leaf nodes of the tree are scalar values.
- Internal nodes are associated with bounding rectangles.
- Bounding rectangles at the same level can overlap.

We first adopted the lock-coupling approach to the quad-B tree. Its implementation is similar to that of an R-tree except the algorithm for a deletion is different. By exploiting the first characteristic given above, the nodes at the same level can be ordered. This has led us to use the link technique in the quad-B tree. The performance of search operations is improved because of less locking overhead.

7.1.2 Rectangular Data

We investigated the R-tree and the quad-R tree for rectangular data. Five different concurrency control approaches were adopted to the R-tree. The R-simple tree is a simple extension to the sequential access method. It uses a single lock to lock the entire tree. When there is an active update operation, no other operation can access the tree. We use the tree as the basis for comparison with other concurrency control approaches. In the second approach, the whole R-tree is locked only when splitting or merging of nodes is required; otherwise, an operation only needs to lock a single node at a time. The second index structure is called the R-lock tree. It improves the level of concurrency by allowing concurrent search and update operations. We further improved the R-tree to the R-couple tree by adopting the

lock-coupling approach. The R-couple tree has a similar behavior to the R-tree for point data. These three trees have been implemented in SR. The simulation results indicate that the R-couple tree has the best search performance when there are many insert operations, while the R-lock tree has the best search performance when there are few insertions.

The preliminary results encouraged us to probe further for better approaches. We then developed the R-opt tree, adopting the give-up approach. It is the first concurrency control protocol we studied, that utilizes both pessimistic (locking) and optimistic (validation) techniques. In an R-opt tree, operations use range information in order to traverse the tree. They can overtake each other and the number of locks needed at each step is small. Besides the three operations described in Chapter 3, a *maintain* operation which reorganizes the R-opt tree has also been developed. However, the overhead in storage can be high and search operations may have to be restarted several times when there are many update operations. The fifth index structure we studied is the R-link tree. In addition to supporting concurrent operations, this tree supports recovery after a system failure. It adopts the link technique and uses pending updates and parent pointers. In the R-link tree, search and insert operations lock only a small number of nodes at each step. However, reorganizing the tree is complex and exclusive locks need to be placed on many nodes.

Using the same idea as in the quad-B tree, we have developed the quad-R tree as the last index structure for rectangular data. For this tree, we used a different method to perform search operations. As in the B⁺-tree, a search window is mapped into query ranges. We suggested to associate a quad-R tree with secondary data structures for better data organization.

7.2 Future Work

A good index structure for spatial data is an integral part of a non-traditional database system. Furthermore, a good index structure should not be limited to supporting sequential accesses, but be able to serve multiple requests at the same time. In the last several years, there has been more and more interest in developing index structures for supporting concurrent access to spatial data [KaF92, ZhD94].

Another important aspect of a spatial database system is recovery after system failures. When a spatial database system crashes due to hardware and/or software failures,

an efficient recovery scheme is needed. The research results about recovery in the traditional database systems may not be directly applicable to a spatial database system due to geometrical relationships among the data.

In this thesis, we have only discussed point and rectangular data. There are other spatial data objects. Moreover, besides the topology of the spatial data, spatial objects may have their own semantic relationships. A spatial index structure that embeds the data semantics as well as supports concurrent access is an interesting research topic in future.

7.2.1 Concurrency Control

Search operations in a spatial database system, such as containment and intersection queries, can use a search window on an index structure to obtain the desired results. The search window query is analogous to the range query to a B^+ -tree which stores only scalar values. In both cases, serializability is often required (see Section 4.1.5 and [GrR92]). Moreover, for spatial data, because it is difficult to have any order among them, an operation may visit leaf nodes in different parts of an index structure. This has limited the direct application of the methods of the B^+ -tree to an index structure for spatial data.

In the thesis, we have used two methods to solve the problem. The first method is to avoid overtaking among operations, but it forces a younger operation not to access any node before an older operation. Hence, the level of concurrency is low. The second method allows overtaking but operations may need to be re-started and there is storage overhead. It will be a challenge to develop a method to resolve all these difficulties.

7.2.2 Recovery

A database system is of little value when recovery of the database is not possible. We have studied the recovery issue in the R-link tree. The tree can be in an incomplete state but still provide correct information. It uses the link pointers to provide alternative routes for tree traversals and atomic actions to reorganize the tree to a complete state slowly. However, because there is no order among the nodes in the tree, it is costly to reorganize it. More work is needed to search for a better method of recovery for a spatial database system.

7.2.3 Other Spatial Data

In [Sam90], Samet describes six different types of spatial data, namely points, lines, rectangles, regions, surfaces, and volumes. This thesis is focused on the point data and rectangular data. Each type of spatial data has its own characteristics and properties. For example, a line segment can be represented by a bounding rectangle of a node in an R-tree or by a number of smaller line segments stored at the terminal quadrants of a quadtree. The choice of a particular representation and the adoption of a concurrency control approach will result in different algorithms. Further work is needed to extend our algorithms or to develop new algorithms for other spatial data.

In a spatial database, a spatial object is frequently a part of another spatial object. Furthermore, there can be several layers of information representing different types of spatial objects. For example, a city block consists of many building lots. When one of the lots is modified, the boundary and/or the size of the city block may be affected. Suppose an R-tree is used for all city blocks in the city and another R-tree for the lots. Then, there is a problem of synchronizing the operations performed on the two trees. If one R-tree is used for both the city blocks and building lots, it may be inefficient when a user is working with city blocks only. Using the quad-R tree, we attempted to solve the problem by partitioning the data space into quadrants and putting them in the leaf nodes. Each quadrant can be used for a particular layer of information. The associated secondary tree of the quadrant can then be used to store the objects in the layer. However, in the quad-R tree, the partitioning is static and the concurrency control protocol does not utilize any semantic information about the data. It will be a future challenge to investigate an index structure which allows dynamic partitioning of the data space and accommodates data semantics in its structure.

Bibliography

- [AbW88] S.K. Abdali and D.S. Wise. Experiments with quadtree representation of matrices, *Proc. of the International Symposium on Symbolic and Algebraic Computation*, Rome, July 1988.
- [AbS83] D.J. Abel and J.L. Smith. A data structure and algorithm based on a linear key for a rectangle retrieval problem, *Computer Vision, Graphics, and Image Processing*, Vol. 24, No. 1, Oct. 1983, pp. 1-13.
- [Abe84] H.J. Abel. A B^+ -tree structure for large quadtrees, *Computer Graphics and Image Processing*, Vol. 27, 1984, pp. 19-31.
- [And82] G.R. Andrews. The distributed programming language SR, *Software-Practice and Experience*, Vol. 12, No. 8, Aug. 1982, pp. 719-754.
- [BaS77] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees, *Acta Inf.*, Vol. 9, 1977, pp. 1-21.
- [BKS90] N. Beckmann, H.P. Kriegel, R. Schneider, B. Seeger. The R^* -tree: An efficient and robust access method for points and rectangles, *Proc. ACM SIGMOD International Conference on Management of Data*, May 23-25, 1990, pp. 322-331.
- [Ben75] J.L. Bentley. Multidimensional binary search trees used for associative searching, *CACM* Vol. 18, No. 9, Sep. 1975, pp. 509-517.
- [Ben77] J.L. Bentley. Algorithms for Klee's rectangle problems, *Technical Report*, Computer Science Department, Carnegie-Mellon University, Pittsburgh.
- [BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*, Addison Wesley, Reading MA (1987).

- [BIM90] H. Blanken, A. Ijbema, P. Meek, and B. Akker. The generalized grid file: Description and performance aspects, *Proc. 6th International Conference on Data Engineering*, Feb. 5-9, 1990, Los Angeles, IEEE, pp. 380-388.
- [Com79] D. Comer. The Ubiquitous B-tree, *ACM Computer Surveys*, Vol. 11, No. 2 (June 1979), pp. 121-138.
- [DRS80] C.R. Dyer, A. Rosenfeld and H. Samet. Region representation: Boundary codes from quadtrees, *CACM*, Vol. 23, 1980, pp. 171-179.
- [Ede80] H. Edelsbrunner. Dynamic rectangle intersection searching, *Institute for Information Processing Rept. 59*, Oct. 1980, Technical University of Graz, Graz, Austria.
- [Ede83] H. Edelsbrunner. A new approach to rectangle intersections, *Intern. J. Computer Math.*, 1983, Vol. 13, pp. 209-219.
- [EdM81] H. Edelsbrunner and H.A. Maurer. On the intersection of orthogonal objects, *Information Processing Letters*, Vol. 13, No. 4,5 (End 1981), pp. 177-181.
- [Ell80] C.S. Ellis. Concurrent search and inserts in 2-3 trees, *Acta Inf.*, Vol. 14, No. 1, 1980, pp. 63-86.
- [Ell82] C.S. Ellis. Extendible hashing for concurrent operations and distributed data, *Proc. ACM SIGMOD International Conference on Management of Data*, 1982, pp. 106-115.
- [Fal86] C. Faloutsos. Multiattribute hashing using Gray codes, *Proc. ACM SIGMOD International Conference on Management of Data*, 1986, pp. 227-238.
- [FSR87] C. Faloutsos, T. Sellis, N. Roussopoulos. Analysis of object oriented spatial access methods, *Proc. ACM SIGMOD International Conference on Management of Data*, 1987, pp. 426-439.
- [FiB79] R.A. Finkel and J.L. Bentley. Quadtrees: a data structure for retrieval on composite keys, *Acta Inf.*, Vol. 4, No. 1, 1979, pp. 1-9.
- [GrR92] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Mateo, CA (1992).

- [Gun87] O. Gunther. Efficient structures for geometric data management, *PhD. dissertation, UCB/ERL M87/77*, Electronics Research Laboratory, College of Engineering, Univ. of California at Berkeley, Berkeley, Calif.
- [GuB91] O. Gunther and J. Bilmes. Tree-based access methods for spatial databases: Implementation and performance evaluation, *IEEE Transactions on Knowledge and Data Engineering, Vol. 3*, No. 3, Sept. 1991, pp. 342-356.
- [Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching, *SIGMOD Record, Vol. 14*, No. 2, June 1984, pp. 47-57.
- [HSW89] A. Henrich, H.W. Six and Peter Widmayer. The LSD tree: spatial access to multidimensional point and non-point objects, *Proc. 15th International Conference on Very Large Databases*, 1989. pp. 45-53.
- [Hin85] K. Hinrichs. The Grid file system: Implementation and case studies of applications, *Doctoral Thesis No. 7734*, ETH Zurich, 1985.
- [HiN83] K. Hinrichs and J. Nievergelt. The grid file: A data structure designed to support proximity queries on spatial objects, *Proc. of the WG'83 (International Workshop on Graphtheoretic Concepts in Computer Science)*, M. Nagl and J. Perl, Eds., pp. 100-113.
- [HuS79] G.M. Hunter and K. Steiglitz. Operations on images using quadtrees, *IEEE Transaction Pattern Anal. Mach. Intell., Vol. 1*, 1979, pp. 145-153.
- [HSW90] A. Hutflesz, H.W. Six and P. Widmayer. The R-file: An efficient access structure for proximity queries, *Proc. 6th International Conference on Data Engineering*, Feb. 5-9, 1990, Los Angeles, IEEE, pp. 372-379.
- [HwD86] C.H. Hwang and W.A. Davis. A file organization scheme for polygon data, *Graphics Interface '86*, pp. 287-292.
- [JoS93] T. Johnson and D. Shasha. The performance of current B-tree algorithms, *ACM Transactions on Database Systems, Vol. 18*, No.1, March 1993, pp. 51-101.
- [JoI84] L. Jones and S.S. Iyengar. Space and time efficient virtual quadtrees. *IEEE Transactions on Pattern Anal. Mach. Intell. Vol. 6*, 1984, pp. 244-247.

- [KaF92] I. Kamel and C. Faloutsos. *Parallel R-trees*, CS-TR-2820, UMIACS-TR-92-1, Computer Science Technical Report Series, University of Maryland, College Park, MD, 1992.
- [Ked83] G. Kedem. The quad-CIF tree: A data structure for hierarchical on-line algorithms, *Proc. of the 19th Design Automation Conference(Las Vegas, June)*, pp. 352-357.
- [KlR79] A. Klinger and M.L. Rhodes. Organization and access of image data by areas, *IEEE Transactions on Pattern Anal. Mach. Intell. Vol. 1*, 1979, pp. 50-60.
- [KoS91] C.P. Kolovson and M. Stonebraker. Segment indexes: Dynamic indexing techniques for multi-dimensional interval data, *Proc. ACM SIGMOD International Conference on Management of Data*, 1991, pp. 138-147.
- [KuL80] H.T. Kung and P.L. Lehman. Concurrent manipulation of binary search trees, *ACM Transactions on Database Systems Vol. 5*, No. 3, 1980, pp. 339-353.
- [KwW82] Y.S. Kwong and D. Wood. Method for concurrency in B-trees, *IEEE Transactions on Software Engineering, Vol. 8*, No. 3, 1982, pp. 211-223.
- [LaS86] V. Lanin and D. Shasha. A symmetric concurrent B-tree algorithm, *Computer Science Dept., New York University Tech. Rep. 210*, April 1986.
- [Lee81] D. T. Lee. Finding intersection of rectangles by range search, *Journal of Algorithms, Vol. 2*, 1981, pp. 337-347.
- [LeY81] P.L. Lehman and S.B. Yao. Efficient locking for concurrent operations on B-Trees, *ACM Transactions on Database Systems, Vol. 6*, No. 4, December 1981, pp. 650-670.
- [LoS90] D.B. Lomet and B. Salzberg. The hB-tree: A multiattribute indexing method with good guaranteed performance, *ACM Transactions on Database Systems, Vol. 15*, No. 4, December 1990, pp. 625-658.
- [LoS91] D. Lomet and B. Salzberg. Concurrency and recovery for index trees, *Technical Report CRL 91/8*, Digital Equipment Corporation, Cambridge Research Laboratory, August 1991.
- [MaL82] U. Manber and R.E. Ladner. Concurrency control in a dynamic search structure, *ACM Proc. on Database Systems(Boston, Mass., April 1982)*, 1982, pp. 268-282.

- [MHN84] T. Matsuyama, L.V. Hao and M Nagao. A file organization for geographic information systems based on spatial proximity, *Computer Vision, Graphics, and Image Processing*, Vol. 26, No. 3, June 1984, pp. 303-318.
- [MiS78] R.E. Miller and L. Snyder. Multiple access to B-trees (preliminary version), *Proc. of the 1978 Conference on Information Sciences and Systems (John Hopkins Univ., Baltimore, MD., 1978)*, John Hopkins Univ., Baltimore, MD., pp. 30-38.
- [McC85] E.M. McCreight. Priority search trees, *SIAM J. Comput.*, Vol. 14, No. 2, May 1985, pp. 257-276.
- [NAO88] Y. Nakamura, S. ABE, Y. Ohsawa, and M. Sakauchi. MD-Tree: A balanced hierarchical data structure for multi-dimensional data with highly efficient dynamic characteristics, *Nineth International Conference on Pattern Recognition*, 1988, pp. 375-378.
- [NgK93] V. Ng and T. Kameda. Concurrent accesses to R-trees, *Proc. 3rd International Symp. on Spatial Databases*, Singapore, June 1993, pp. 142-161.
- [Ng94a] V. Ng and T. Kameda. The R-Link tree. *CSS/LCCR TR94-08*, August 1994, School of Computing Science, Simon Fraser University, Burnaby, Canada.
- [Ng94b] V. Ng and T. Kameda. The R-Link tree: A recoverable index structure for spatial data, *Proc. 5th International Conference on Database and Expert Systems Applications*, September 7-9, 1994, Athens, Greece.
- [NgK95] V. Ng and T. Kameda. Concurrent access of point data, *Submitted to Data Engineering'95*.
- [NHS84] J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The grid file: An adaptable, symmetric multikey file structure, *ACM Transactions on Database Systems*, Vol. 9, No. 1, March 1984, pp. 38-71.
- [OrM84] J.A. Orenstein and T. Merrett. A class of data structures for associative searching, *Proc. SIGMOD-SIGACT PODS, Symp. Principles of Database Systems*, 1984, pp. 181-190.
- [Oto90] E.J. Otoo. Efficient concurrency control protocols for B-tree indexes, *SCS-TR-166*, Jan. 1990, School of Computer Science, Carleton University, Ottawa, Canada.

- [Rob81] John T. Robinson. *The K-D-B-Tree: A search structure for large multidimensional dynamic indexes*, *Proc. ACM SIGMOD International Conference on Management of Data*, 1981, pp. 10-18.
- [RoL85] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees, *Proc. ACM SIGMOD International Conference on Management of Data*, 1985, pp. 17-31.
- [Sam80] H. Samet. Region representation: Quadtrees from boundary codes, *CACM Vol. 23*, 1980, pp. 163-170.
- [Sam82] H. Samet. Neighbor finding techniques for images represented as quadtrees, *Computer Graphics and Image Processing, Vol. 18*, 1982, pp. 37-57.
- [Sam88] H. Samet. Hierarchical representations of collections of small rectangles, *ACM Computing Surveys, Vol. 20*, No. 4 (December 1988), pp. 271-309.
- [Sam90] H. Samet. *The Design and Analysis of Spatial Data Structures*, Addison-Wesley Series in Computer Science, 1990.
- [Sag86] Y. Sagiv. Concurrent operations on B^* -trees with overtaking, *Journal of Computer and System Sciences, Vol. 33*, pp. 275-296 (1986).
- [SeK88] B. Seeger and H.P. Kriegel. Techniques for design and implementation of efficient spatial access methods, *Proc. 14th VLDB Conference*, 1988, pp. 360-371.
- [SRF87] T. Sellis, N. Roussopoulos, C. Faloutsos. The R^+ tree: A dynamic index for multi-dimensional objects, *Proc. 13th VLDB Conference*, Brigton, 1987.
- [Sha86] C.A. Shaffer. Application of alternative quadtree representations, *Ph.D. dissertation, TR-1672*, Computer Science Department, University of Maryland, College Park, MD, June 1986.
- [ShG88] D. Shasha and N. Goodman. Concurrent search structure algorithms, *ACM Transactions on Database Systems, Vol. 13*, No.1, March 1988, pp. 53-90.
- [Slo81] K.R. Sloan Jr.. Dynamically quantized pyramids, *Proc. 6th International Joint Conference on Artificial Intelligence, Vancouver*, August 1981, pp. 734-736.

- [SrC91] V. Srinivasan and M.J. Carey. Performance of B-tree concurrency control algorithms, *Technical Report, Computer Science Department*, University of Madison, 1991.
- [Tam81] M. Tamminen. *The EXCELL method for efficient geometric access to data*, Acta Polytechnica Scandinavica, Mathematics and Computer Science Series No. 34, Helsinki, Finland 1981.
- [ZhD94] X.Y. Zhou and W.A. Davis *A Semantic Approach for Concurrency in R-trees*, Computer Science Technical Report, University of Alberta, 1994.