THE INDEX BY DIMENSIONAL PROJECTION

— AN INDEX SUPPORTING SEARCH FOR

SPATIAL OBJECTS BY REGION

BY

XIAOMING CHENG

Bachelor of Science

Shanghai Jiao Tong University

Shanghai, China

1982

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the Degree of
MASTER OF SCIENCE
May, 1991

THE INDEX BY DIMENSIONAL PROJECTION

— AN INDEX SUPPORTING SEARCH FOR

SPATIAL OBJECTS BY REGION

Thesis Approved:

Huizhu Lu

Thesis Adviser

G. E. Hedrick

J Chandler

Norman N Durham

Dean of the Graduate College

PREFACE

In image databases and other spatial data retrieval systems, the techniques for storing and indexing data objects require different kinds of search and query from those in traditional databases and data retrieval systems. In order to handle spatial data more efficiently, a new index structure supporting search for spatial objects by region, the Index by Dimensional Projection is proposed in this thesis. By this method, the number of pages accessed for searching a point region has a logarithmic relationship with the number of objects in data space and the number of comparisons required for searching an entry within a disk page has logarithmic relationship with the number of entries in the disk page.

I wish to express my sincere gratitude to the individuals who assisted me in this project and during my coursework at Oklahoma State University. In particular, I wish to thank my major adviser, Dr. Huizhu Lu, for her guidance on this study. I am also grateful to other committee members, Dr. George E. Hedrick and Dr. John P. Chandler, for their advisement during the course of this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

NOMENCLATURE

[ )	semi-open segment or semi-open rectangle
C	capacity of a page of an IDP
Cr	capacity of a page of an R-tree or an R^+-tree
D	density of objects in a data space
D_i	density of objects in a slice of a data space in the ith dimension
f	fan-out of a page of an IDP
f r	fan-out of a page of an R-tree or an R^+-tree
G	starting point of the segment in an entry of an IP-tree or a TP-tree
Hi	height of an IP-tree
Ht	height of a TP-tree
lb_i	lower boundary of a rectangle in the ith dimension
m	lower boundary of the number of entries in a page
M	upper boundary of the number of entries in a page

n	number of dimension of a data space
N	number of objects in a data space
N_i	number of objects in a slice of data space in the ith dimension
P	pointer in an entry
u	number of MCCSs resulting from a projection
ub_i	upper boundary of a rectangle in the ith dimension
u_i	number of MCCSs resulting from the projection of a sub^{i-1}-space onto the ith axis
w_i	width of a rectangle along the ith dimension

GLOSSARY

capacity: The number of entries a leaf node can have.

cover: An n-d rectangle, R1, covers another n-d rectangle, R2 if every point contained by R2 is contained by R1 also. A segment, S1, covers another segment, S2 if every point contained by S2 is contained by S1 also.

disjoint: Two n-d rectangles or two segments disjoint from each other if there is no point contained by the both rectangles or the both segments.

ECP (Equally Covered Points): ECP are points on an axis and covered by the same set of projections of rectangles onto the axis.

enclosing rectangle: The enclosing rectangle of an object or some other enclosing rectangles is the smallest rectangle which covers the object or those rectangles.

end point: For a segment represented by [sl, su), the end point is the poin at su.

Equally Covered Points: (see ECP)

fan-out: The number of sub-trees a non-leaf node can have.

IDP (Index by dimensional Projection): The IDP is an index structure supporting search spatial objects by region. An IDP indexes the spatial objects by projecting them in each dimension.

Index by dimensional Projection: (see IDP)

Intermediate Projection tree: An Intermediate Projection tree is a part of an IDP of an n-dimensional data space. An intermediate Projection tree indexes a sub-space by the projection of the sub-space onto an axis other than the axis in the nth dimension.

intersect: Two n-dimensional rectangles or two segments intersect each other if there are at least one point contained by the both rectangles or the both segments.

IP-tree: (see Intermediate Projection tree)

Maximum Constantly Covered Rectangle (MCCR): A Maximum Constantly Covered Rectangle is a rectangle in which every point is contained by the same set of objects, and any rectangle which contains the rectangle and is not equal to the rectangle contains at least two points contained by different set of objects.

Maximum Constantly Covered Segment (MCCS): A Maximum Constantly Covered Segment is a segment on an axis, in which all points are ECP, and any segment which contains the segment and is not equal to the segment contains at least two points which are not ECP.

MCCR: (see Maximum Constantly Covered Rectangle)

MCCS: (see Maximum Constantly covered Segment)

overlap: (see intersect)

point query (point search): An operation to determine all the objects containing a certain point.

point search: (see point query)

R-tree: The R-tree is an extension of the B-tree without the property of key ordering. The R-tree supports search spatial objects by region.

R^+-tree: The R^+-tree is a variation of the R-tree with the restriction of not allowing the overlap among nodes on the same level.

region: An rectangular space within a data space.

search by region: An operation to determine all the objects intersecting a given region.

sequential list: A sorted linked list of all leaf nodes in an IP-tree or a TP-tree, like that in a B^+-tree.

spatial data object: A data object covering intervals in each dimension of the data space.

stage: The step of projection to index spatial objects by the IDP. Each stage corresponds to a step of projection of the spatial objects onto a corresponding axis.

starting point: For a segment represented by [sl, su), the starting point is the point with the coordinate value of sl. For an n-d rectangles represented by [sl_1, su_1, sl_2, su_2, ..., sl_n, su_n), the starting point is the point with the coordinate of [sl_1, sl_2, ..., sl_n).

sub-space: A sub-space is a part of the data space formed by dividing the space by MCCSs generated from each projection of the data space.

Terminal Projection tree (TP-tree): A Terminal Projection tree is a part of an IDP. A terminal projection tree indexes the intersections of objects with the MCCRs generated from a projection of the nth stage.

TP-tree: (see Terminal Projection tree)

uniform list: A list of consecutive entries stored in the sequential list of a TP-tree and corresponding to the objects intersecting an MCCR.

CHAPTER I

INTRODUCTION

Requirement

In most programming languages and database systems, both numeric and string data types are available. With the growth of the requirements of describing and manipulating spatial data objects, such as in image databases and CAD, much work has been done in the fields of image data definition, representation and processing. However, comparatively less work has been done to meet the requirements of organizing spatial data objects in large collections and to support storing and retrieving these objects efficiently.

In traditional databases and data retrieval systems, many well-developed techniques can be used to support data storage and retrieval. There are many kinds of data storing structures and index structures for alphanumeric data types. In image databases and other spatial data retrieval systems, the techniques for storing and indexing data objects require different kinds of search and query to improve the efficiency. However, there are not many choices [Same90].

Among all kinds of search or query operations for spatial data objects, search by region is one of the most different manipulations

from those for alphanumeric data types. The traditional alphanumeric data always are points in a data space, but spatial data objects are intervals in data spaces and cover areas. For example, temporal data can be viewed as one-dimensional spatial data with intervals between beginning and end points; geographic applications and VLSI design involve two-dimensional data; geological data and solid modelling applications require three-dimensional data and sometimes require four-dimensional data. Queries for these kinds of data objects often are related to the areas they are covering.

Query by region, which is an operation to determine all data objects intersecting a given region, is embedded in some query languages of pictorial database systems such as PSQL (**P**ictorial **S**tructured **Q**uery **L**anguage). PSQL is a relational based language for retrieving information from a pictorial database. It extends the power of SQL (**S**tructured **Q**uery **L**anguage) for retrieving alphanumeric data by allowing direct spatial search. The pictorial database maintains the associations between the spatial and alphanumeric objects.

PSQL extend mapping is in the form:

select <attribute_target_list>
from <relation_list>
on <picture_list>
at <area_specification>
where <qualification>;

The following example is a typical simple query.

```
select    city, state,population
  from    cities
    on    US_map
      at    loc  overlapping { 4±4,11±9 }
        where    population > 500000;
```

which select all cities in the area {4±4,11±9} having population greater than 500,000.

If there is no index supporting the search of spatial objects by region, then the search for the cities within a given region on the picture_list has to be implemented in a method such as

```
R = {4±4, 11±9};
for each CITY on US_map
  if overlap(CITY, R)
    output(CITY);
```

The search method of "for each" requires a linear search and is performed slowly.

Problems

In order to handle spatial data efficiently, as required in both computer-aided design and geometric data applications, a database system should have an index mechanism to retrieve data items quickly according to their spatial locations. However, traditional indexing methods are not well suited to data objects with non-zero size located in multidimensional space.

Two existing index structures, the R-tree [Gutt84] and the R^+-tree [Sell 87], were introduced to meet these needs. Both R-trees and R^+-trees are extensions of B-trees. Both R-trees and R^+-trees maintain their balanced heights as well as the property of logarithmic page access.

Although the R^+-tree improves the performance of page access in a search by eliminating the overlap among nodes on the same level, which exists in the R-tree, algorithms for insertion and deletion in an R^+-tree are more complicated and have greater complexity than those for an R-tree. Moreover, the performance of searching in both the R-tree and the R^+-tree might be diminished in dynamic circumstances if the optimization of the organization of these trees either is not included in the insertion and/or deletion algorithms or is not applied to these trees periodically.

Solution

In order to handle spatial data more efficiently, a new index structure, IDP which stands for **I**ndex by **D**imensional **P**rojection is proposed in this thesis. An IDP is a cluster of extended B^+-trees.

An IDP has the following advantages:

(1) The performance of page access in searching in an IDP is better than that in an R-tree and similar to that in an R^+-tree, and even better when page size is small.

(2) The algorithmic complexity for searching for an entry in a node of an IDP is less than it is in a node of either an R-tree or an

R^+-tree.

(3) The optimization of organizing an IDP has a linear complexity which is lower than the optimization of organizing either an R-tree or an R^+-tree.

Terminology

Before examining previous work related to index structures, the following terminology must be understood.

n-dimensional space:

An n-dimensional (denoted by n-d in the rest of the thesis) space is an n-dimensional Cartesian coordinate system with upper and lower bounds for each dimension and with discrete grid of coordinates.

Segment:

A segment [sl, su) covers the area between sl and su in a 1-dimensional (1-d) space. sl and su are the lower and upper boundaries of a segment. A segment is semi-open: sl is contained in the segment but su is not.

Unit segment:

A unit segment is the minimal distinguishable segment in a digital space. A unit segment is indivisible. The size of a unit segment depends on the resolution of the space. A 1-d space is covered by all unit segments in the space. Any two unit segments are disjoint from each other.

n-dimensional rectangle (n-d rectangle):

An n-dimensional rectangle is a rectangle perpendicular to the coordinate system. That is, every edge of the rectangle is parallel to its corresponding coordinate axis. Each edge of an n-dimensional rectangle is a segment. So an n-dimensional rectangle is semi-open. An rectangle is described in the form of

$$RECT = [lb_1, ub_1, lb_2, ub_2, ..., lb_n, ub_n)$$

where lb_i and ub_i are the lower and the upper bounds of the rectangle in the ith dimension, correspondingly. The interval between two boundaries is semi-open. That is, lb_i is included in the interval but ub_i is not. n is used as the notation of the number of the dimensions of the data space in the rest of the thesis.

Region:

A region is an n-d rectangle in an n-d space.

Unit region:

A unit region is also called a point region. A unit region is a region with the smallest area the digital space can represent. Each edge of a unit region is a unit segment. Like unit segments, unit region are indivisible and disjoint. An n-d space is covered by all its unit regions. The size or the area of a unit region depends on the resolution of the digital space.

Zero-size point:

A zero-size point does not cover any area. It only has its position. Every zero-size point in the space is contained exactly by one unit region. In the digital space, different zero-size points in the same unit region are undistinguishable. Zero-size points in

different unit regions are distinguished by distinguishing the unit regions.

Spatial object:

A spatial object is a tuple (or a record) representing an object in an n-d space, such as a tuple for a city on a map, for a shaft in a gear box or for a piece of connection on a layout of a VLSI. Besides the attributes contained in a tuple for an alphanumeric object, a tuple for a spatial object must contain the position of the object and a representation of the n-d area the object covers.

Enclosing rectangle:

An n-d enclosing rectangle is an enclosing rectangle either of a spatial object or of some other enclosing rectangles. An enclosing rectangle of a spatial object is the minimum n-d rectangle containing the object. Likewise, an enclosing rectangle of some other enclosing rectangles is the minimum rectangle containing those rectangles.

Search by region:

Search by region determines all spatial objects intersecting a given region (also called *search region* or *query window*). The given region is usually an n-d rectangle. If the search region is a point region, the search is called *point query* or *point search* which is to find out all spatial objects containing this point.

Because a search region is a rectangle, it can be described in the same form of an enclosing rectangle.

For simplicity, the area covered by a spatial object is

considered as an n-d rectangle in indexes. A spatial object is stored in indexes in the form of

(OID, RECT)

where OID is the identifier of the spatial object enclosed by the enclosing rectangle, RECT.

CHAPTER II

REVIEW OF INDEX STRUCTURES

Traditional Index Structures

Traditional data processing has dealt only with alphanumeric data types (i.e. numerals and strings) and with either numeric or string computations. Since database systems emerged from the same environment, their data types are also limited to alphanumeric types. So data management is limited to structured collections of alphanumeric values.

New requirements came with the introduction of pictorial databases. Chang provided an survey of most of the attempts in the area of pictorial databases [Chan81a]. Some of the classical database techniques were extended in several respects to meet the pictorial requirements [Chan81b][Chan81c].

In pictorial databases, data objects are different from those in traditional databases. In traditional databases, a data object is a data point and can be represented by a vector. In pictorial databases, data objects are spatial. Manipulations of spatial data objects such as spatial search and spatial computations must be implemented with the support of proper data structures.

Spatial data objects often cover areas in multi-dimensional

spaces, not well-represented by point locations. For example, map objects such as counties, lakes, cities etc. occupy regions of non-zero size in two dimensions. A common operation on spatial data is a search for all objects in an area, the search by region, such as to implement the **on** clause and the **at** clause in a query in PSQL. This kind of spatial search occurs frequently in computer-aided design (CAD) and geo-data applications. Therefore, it is important to retrieve objects efficiently according to their spatial locations.

One Dimensional Index Structures

Traditional one-d index structures are not appropriate for performing a multi-dimensional spatial search. Structures based on exact matching of values, such as hash tables are not useful because a range search is required. Mapped by a hash function, the spatial positions of data objects no longer exist in a hash table. Structures using one-dimensional ordering of key values, such as sorted linear table, binary search trees, B-trees and the variations to binary trees [Knut73] and to B-trees [Come79], do not work because the search space is multi-dimensional.

A data object with more than one key can be indexed with a one-dimensional index if all its keys are concatenated into one. A concatenation of keys is equivalent to an arrangement of keys with the choice of one key to be the primary key and the others to be auxiliary in some order. In this way, a data object is considered as one-dimensional although it has more than one key.

Multi-Dimensional Index Structures

The multi-dimensional binary search tree (k-d-tree) was introduced by Bentley [Bent75]. The k-d-tree is a natural generalization of the binary search tree to handle the case of a single record having multiple keys. Every key of a data record matches the dimensional position of the record in a k-dimensional space. Each key of a record is considered as the value of the corresponding coordinate. The multi-dimensional binary search tree supports associative searching, thereby dealing with a multiplicity of keys. In 1979, Bentley discussed the variations of k-d-trees and the applications of different kinds of associative searches on k-d-trees in database applications [Bent79].

Since a k-d-tree can be very large as an index for many records, an implementation of a k-d-tree on a secondary storage device is necessary. In order to reduce disk page access, the method of storing "close" nodes on the same disk page [Knut73] is recommended for organizing a k-d-tree on secondary memory [Bent79]. Multi-dimensional data objects can be indexed in the efficient and simple data structure of a k-d-tree.

In 1981, a multi-dimensional B-tree (k-d-B-tree) was introduced [Robi81]. The k-d-B-tree supports both an exact matching search and a range search of multi-dimensional data objects. Like the B-tree, the k-d-B-tree maintains its height balance and low number of page accesses on disk. Both k-d-tree and the k-d-B-tree are indexes for point data objects in multi-

dimensional space.

R-tree

In 1984, Guttman introduced an index structure, the R-tree [Gutt84], to support search by region for non-atomic, non-zero size, spatial objects. Search for spatial objects by region can be performed directly on this kind of index.

An R-tree is a height balanced tree. A leaf node in an R-tree contains entries of the form

(OID, RECT)

where OID refers to the tuple of a spatial object and RECT is the n-d rectangle bounding the object, the enclosing rectangle. A non-leaf node contains entries of the form

(ptr, RECT)

where ptr is the address of a lower node in the R-tree and RECT is the enclosing rectangle of all the rectangles of the entries in the lower node. An R-tree has the following properties:

(1) Every node is stored on one disk page and has at most M entries. M is determined by the sizes of the disk page and the entry.

(2) Every leaf node contains between m and M entries unless it is the root. Usually, $m \leq M/2$.

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

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

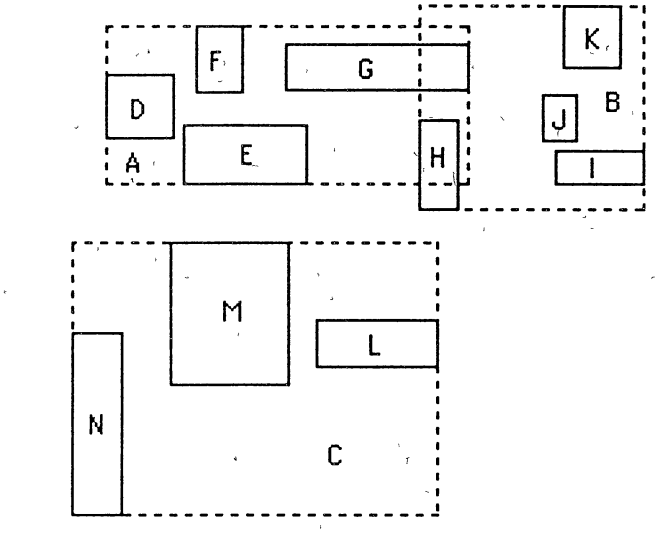
(5) All leaves are on the same level.

Figure 1 is an example of R-tree.

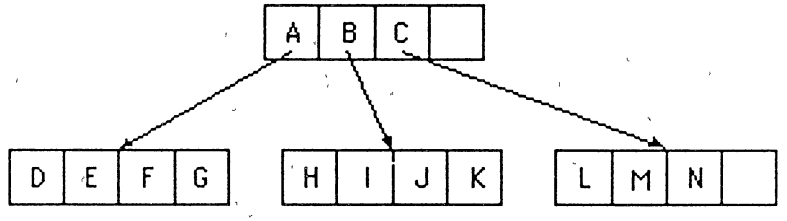
A search by region can be performed by the algorithm R_SEARCH in Figure 2.

The R-tree structure does not restrict overlap among rectangles of entries on the same level. When a search region is covered by more than one non-leaf node, all sub-trees rooted at these nodes must be searched. For example, when the search region is W (Figure 3), both sub-tree A and B must be searched although no object in A overlaps W. In some cases, even a point search requires the searching of several sub-trees.

When an R-tree is applied in a dynamic circumstance, the rectangles of objects are inserted and deleted frequently. Therefore, the organization of the R-tree is changed in every insertion or deletion. Different methods of grouping the rectangles in a R-tree results in different organizations of the R-tree. Figure 4 is another example of organization for the set of rectangles in Figure 1. The shaded areas are the overlapping of the rectangles in the same node. The more overlapping area is in an R-tree the more page access is possibly required in a search operation. For searching the R-tree in Figure 4 by the same region W as in Figure 1 , one more page has to be accessed. Because the organization of R-trees is important to the performance of R-tree, Roussopoulos and Leifker [Rous85], in 1985, presented an initial packing technique for improving the organization of objects in an R-tree to reduce the



(a)



(b)

Figure 1. An R-tree

(a) 2-dimensional rectangles organized in an R-tree.
(b) The corresponding R-tree.

Algorithm R_SEARCH (NODE, W)

Collect all objects intersecting the query window W in an R-tree rooted in NODE.

S1: If NODE is not a leaf, then for each entry (ptr, RECT) in NODE check whether RECT intersects W. If so, SEARCH(ptr, W).

S2: If NODE is a leaf, check all objects in NODE, and return the objects which intersect W.

Figure 2. Algorithm R_SEARCH

number of page accesses in the search algorithm.

R^+-tree

In 1987, the R^+-tree, a variation of the R-tree, was introduced by Sellis, et al [Sell87]. The R^+-tree has a performance of page access when searching that is better than that of the R-tree. R^+-trees allow no overlap among rectangles of nodes at the same level (Figure 5). The entries in leaf nodes and non-leaf nodes in an R^+-tree have the same forms as those in an R-tree. An R^+-tree has the following properties:

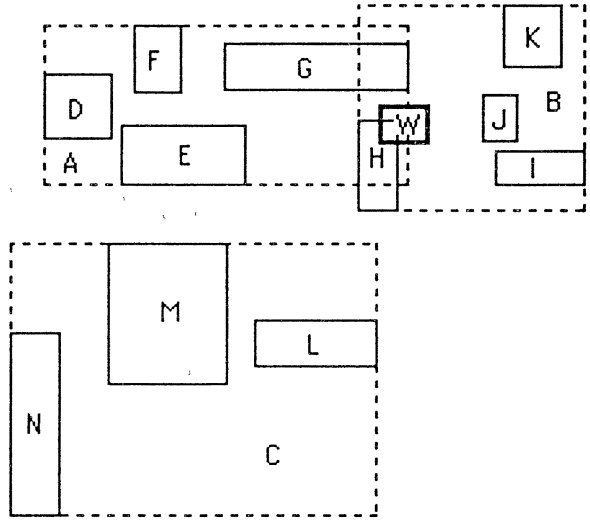
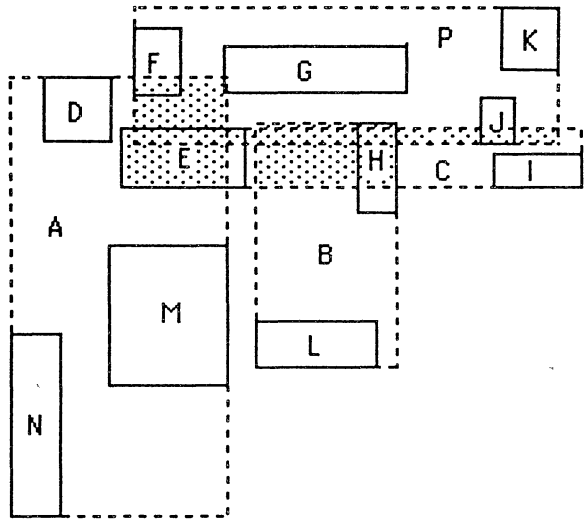
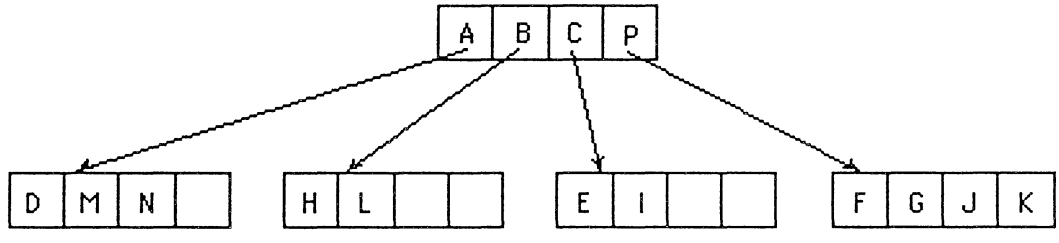


Figure 3. A Search Window in an R-tree

Search window W intersecting both rectangle A and B.



(a)



(b)

Figure 4. A Poorly Organized R-tree

(a) 2-dimensional rectangles organized in an R-tree.
(b) The corresponding R-tree.

(1) For each entry (ptr, RECT) in a non-leaf, the sub-tree rooted at the node pointed to by ptr contains rectangle R if and only if R is covered by RECT. The only exception is when R is a rectangle at a leaf node. In that case, R must just overlap with RECT.
(2) For any two entries (ptr1, RECT1) and (ptr2, RECT2) of an non-leaf node, the overlap between RECT1 and RECT2 is zero.
(3) The root has at least two children unless it is a leaf.
(4) All leaves are on the same level.

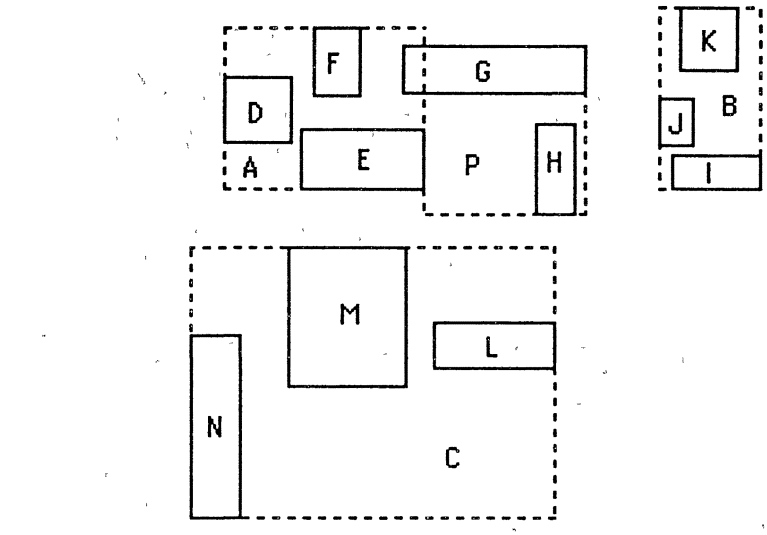
The algorithm R_SEARCH in Figure 2 can be applied directly to search in an R^+-tree by region.

Searching on R^+-tree has a page access complexity lower than on an R-tree. But, to organize an R^+-tree requires more complicated algorithms of dividing and grouping rectangles. As in R-trees, a poorly organized R^+-tree results in more page accesses for the searching algorithm. Figure 6 shows an R^+-tree indexing for the same set of rectangles as in the Figure 5. The height of the R^+-tree in Figure 6 is greater than in Figure 5 because of a poor organization of the tree.

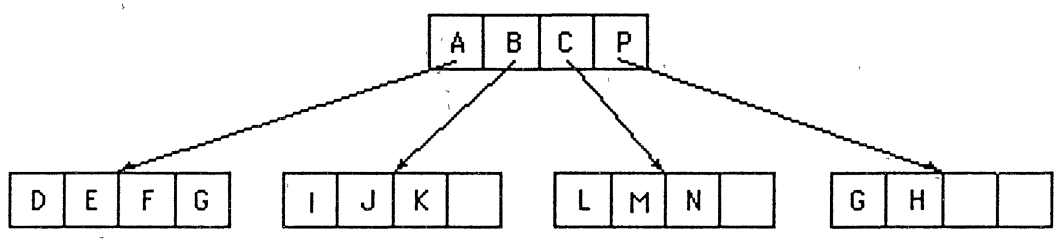
In 1989, a variation of the R^+-tree, the Cell tree, was introduced by Gunther [Gunt89]. The structure of the Cell tree is similar to that of the R^+-tree but the manner of enclosing data objects and grouping entries into nodes is different. Instead of using the minimum rectangles in the R^+-tree, the Cell tree uses convex to enclose data objects and to divide and cover data space.

The building of an optimally organized R-tree or R^+-tree requires a combinational algorithm. To make a well-organized R-tree or R^+-tree in a tractable method, some algorithms to build the semi-optimal R-tree or R^+-tree were introduced [Gutt84][Rous85][Sell87]. All of these algorithms require a large amount of running time and a large number of page accesses.

The performances of search algorithms for both R-trees and R^+-trees were analyzed and compared in 1987 [Falo87a][Falo87b]. Roussopoulos et al proposed R^+-tree to be the index in Pictorial Structured Query Language (PSQL) to support a direct spatial search [Rous88]. In 1989, Goodman et al discussed the work to invent new languages and to extend existing languages in semantics for knowledge-based computer vision management systems [Good89]. In their paper, R-tree nodes are used as the type of data records to index the topology fields on the bounding boxes on a logarithmic time. However, just as Grosky and Mehrotra said, "Except for the design of R-trees ... , database designers have not concentrated on designing efficient access methods for image databases [Gros89]."



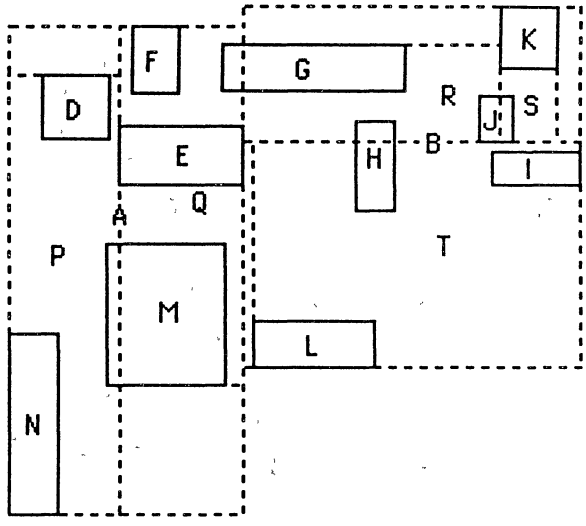
(a)



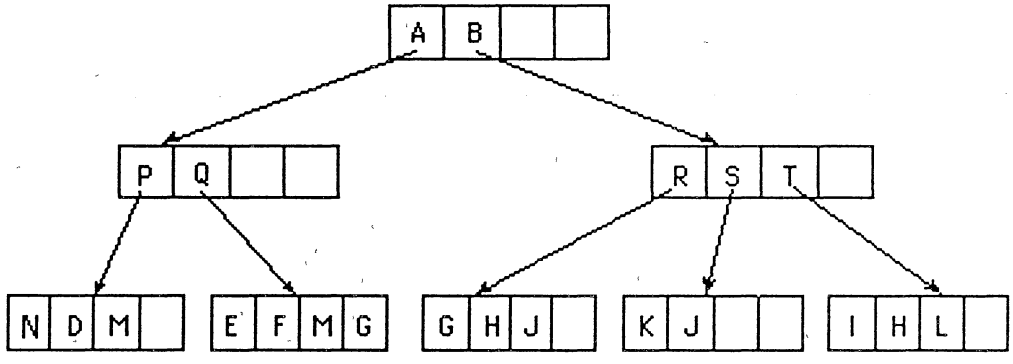
(b)

Figure 5. An R+-tree

(a) 2-dimensional rectangles organized in an R+-tree.
(b) The corresponding R+-tree.



(a)



(b)

Figure 6. A Poorly organized R+-tree

(a) 2-dimensional rectangles organized in an R+-tree.
(b) The corresponding R+-tree.

CHAPTER III

INDEX BY DIMENSIONAL PROJECTION

Introduction

The existing index structures supporting direct search spatial objects by region, R-trees and R^+-trees, have some disadvantages:

(1) R-trees do not restrict the overlap among rectangles of nodes on the same level. When a search region is covered by more than one non-leaf node, all the sub-trees rooted at these nodes must be accessed. Search efficiency degrades.

(2) R^+-trees allow no overlap among rectangles of nodes on the same level. Although search efficiency is higher than searches of R-trees, the algorithms for splitting and grouping rectangles while organizing an R^+-tree are much more complicated as well as more complex.

(3) Entries in the node of either an R-tree or an R^+-tree are unsorted. Search of an entry in such a node requires a linear search.

(4) In dynamic circumstances, the search efficiencies of either an R-tree or an R^+-tree might degrade seriously if no optimization is applied. The optimization for an R-tree or for an R^+-tree requires combinational operations and is an NP problem.

The proposed index structure for spatial objects, Index by

Dimensional Projection (IDP), has a performance of page access when searching that is better than that of the R-tree and similar to that of the R+-tree and even better in the cases of small page sizes. Its performance in terms of time for searching an entry within a page is much better than those of the R-tree and the R+-tree.

Terminology

A Boolean function, COVER(arg_1,arg_2) is true if both arg_1 and arg_2 are n-d rectangles and arg_2 is included in arg_1, or, arg_1 is an n-d rectangle and arg_2 is a segment (or a point, a zero-size segment) on the ith axis and arg_2 is included in the projection of arg_1 onto the ith axis, for $0 < i \leq n$, otherwise COVER(arg_1, arg_2) is false.

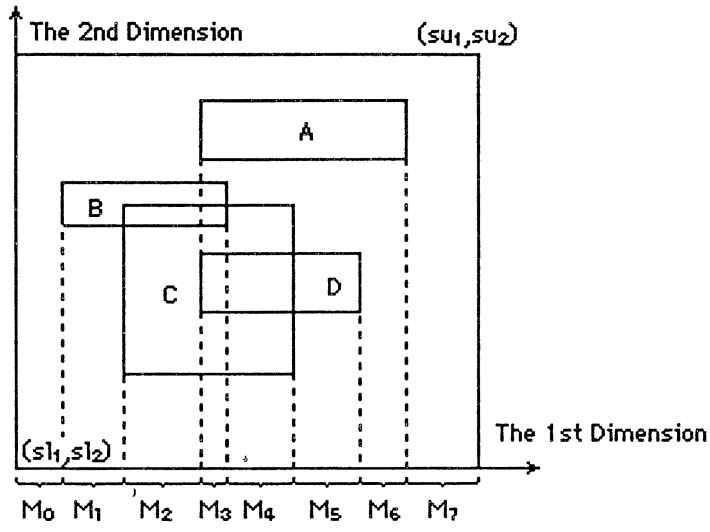
A Boolean function INTERSECT(arg_1,arg_2) is true if both arg_1 and arg_2 are n-d rectangles and there is at least one point within both arg_1 and arg_2, or, arg_1 is an n-d rectangle and arg_2 is a segment on the ith axis, and there is at least one point in both arg_2 and also in the projection of arg_1 onto the ith axis, for $0 < i \leq n$, otherwise, INTERSECT(arg_1, arg_2) is false.

Points, p_1 and p_2, on the ith axis are **E**qually **C**overed **P**oints (**ECP**) if for every rectangle R in the data space, either COVER(R, p_1) and COVER(R, p_2) are both true or are both false.

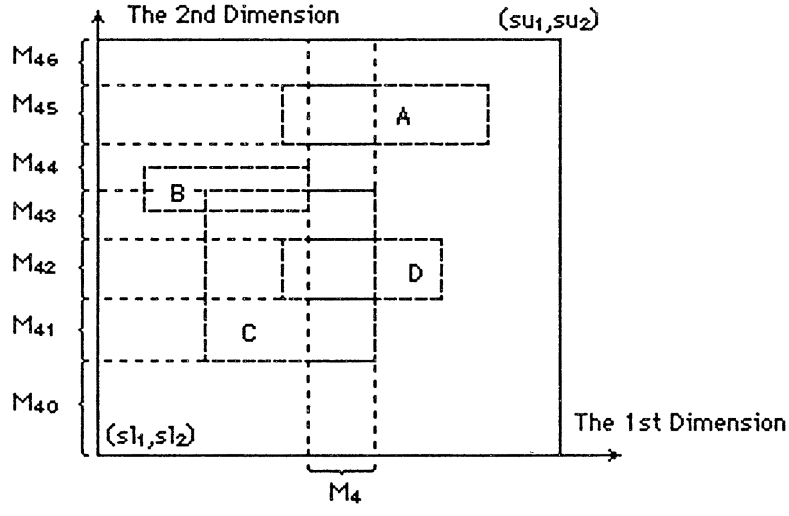
A segment, S, on the ith axis is a **M**aximum **C**onstantly **C**overed **S**egment (**MCCS**), if any pair of points p_1 and p_2 on S are ECP, and, there exists no such a segment S_1 adjacent to S that every pair of points p_3, on S, and p_4, on S_1, are ECP. An MCCS is the largest

segment which covers consecutive ECP. MCCSs on an axis are disjoint and consecutive. In Figure 7(a), the four 2-d objects: A, B, C and D, is projected to the 1st axis. This projection results in 8 MCCSs, from M_0 to M_7

If the rectangles in an n-d data space, [lb_1, ub_1, lb_2, ub_2, ..., lb_n, ub_n), are projected onto the axis of the first dimension, then the axis is divided into MCCSs. If the number of the MCCSs is k and each of these MCCSs is represented by a segment [Ml_i, Mu_i), $1 \leq i \leq k$, then the data space can be divided into k sub-spaces. Each of these sub-spaces is occupied by the rectangle [Ml_i, Mu_i, lb_2, ub_2, ..., lb_n, ub_n), for $1 \leq i \leq k$, correspondingly. For example, there are 8 sub-spaces, [Ml_k, Mu_k, lb_2,ub_2), for $0 \leq k \leq 7$, in Figure 7(a). These sub-spaces are said to be 1-order sub-spaces denoted by sub^1-spaces, because they are generated from dividing an original data space by the MCCSs on the 1st axis. A sub^j-space can be divided into sub^{j+1}-space by the MCCSs which are generated from the projection of the sub^j-space with all rectangles intersecting it onto the (j+1)st axis, $1 \leq j < n$. For example, one of the sub^1-space in Figure 7(a), M_4=[Ml_4, Mu_4, lb_2,ub_2), is further projected onto the 2nd axis. Then, 7 sub^2-spaces are generated as shown in FIgure 7(b), each of which is [Ml_4,Mu_4, Ml_{4k}, Mu_{4k}), for $0 \leq k \leq 6$, on the assumption that the MCCS M_{4k} is [Ml_{4k}, Mu_{4k}). The numbering method for axes in this thesis is based on the ordering of projections, thus,each MCCS on the ith axis corresponds to a sub^i-space. The whole data space is the sub^0-space. After n steps of divisions, all the resulting sub^n-spaces are called **M**aximum



(a)



(b)

Figure 7. The Projection of a 2-dimensional Space

(a) The projection onto the first axis.
(b) The projection of a sub1-space onto the second axis.

Constantly **C**overed **R**ectangles (**MCCR**). The 7 sub^2-spaces, [Ml_4,Mu_4, Ml_{4k}, Mu_{4k}), for $0 \leq k \leq 6$ (see Figure 7(b)), are MCCRs because n=2.

Structure

An **IDP** consists of n-1 stages (from the 1st stage through the (n-1)st stage) of **I**ntermediate **P**rojection trees **(IP-trees)** and one stage (the nth stage) of **T**erminal **P**rojection trees **(TP-trees)**.

An IP-tree on the ith stage indexes the MCCSs generated from the projection of the rectangles intersecting a sub^{i-1}-space onto the ith axis, for $1 \leq i < n$. Because an MCCS on the ith axis is corresponding to a sub^i-space, the IP-tree on the ith stage is the index for these sub^i-spaces. The IP-tree in Figure 8(a) corresponds to the index for the MCCSs resulting from the projection of data space (the sub^0-space) onto the first axis as shown in Figure 7(a).

Each of these sub^i-spaces is projected further onto the axis of the (i+1)st dimension, then each of these projections results in a new set of sub^{i+1}-spaces. Each set of the sub^{i+1}-spaces from a projection of a sub^i-space are indexed by a corresponding IP-tree on the (i+1)st stage. The generated sub^{i+1}-spaces are divided by the projection onto the next axis and indexed by the IP-trees on the next stage, and so on, until the sub^{n-2}-spaces are projected onto the (n-1)st axis. Then the sub^{n-1}-spaces are projected on the nth axis and divided into MCCRs. Each of these projections onto the nth axis

corresponds to a TP-tree. A TP-tree indexes all the intersections of objects with the MCCRs. In Figure 7(b), the sub^1-space corresponding to M_4, [Ml_4, Mu_4,lb_2, ub_2) in Figure 7(a) is projected on the 2nd axis (the nth axis in the 2-d space). This projection results in seven MCCRs. The intersections of the objects with these MCCRs are indexed by the TP-tree in Figure 8(b). Because the TP-tree in Figure 8(b) is the index for the further projection of the sub^1-space corresponding to M_4, the TP-tree is pointed to by the entry of M_4 in the leaf node of the IP-tree in Figure 8(a). The relationship of the IP-trees and the TP-trees are shown in Figure 8(c).

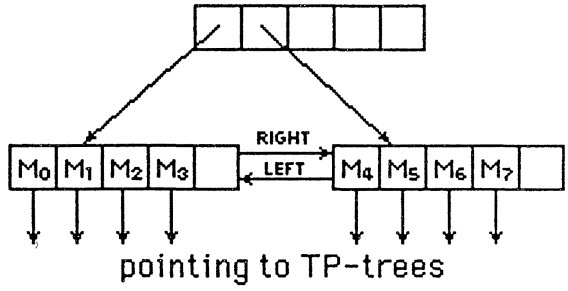
IP-tree

An IP-tree is an instance of a B$^+$-tree [Come79][Knut73]. An IP-tree on the ith stage corresponds to the projection of a sub^{i-1}-space onto the ith axis and indexes the MCCSs result from this projection. An IP-tree has the following properties.

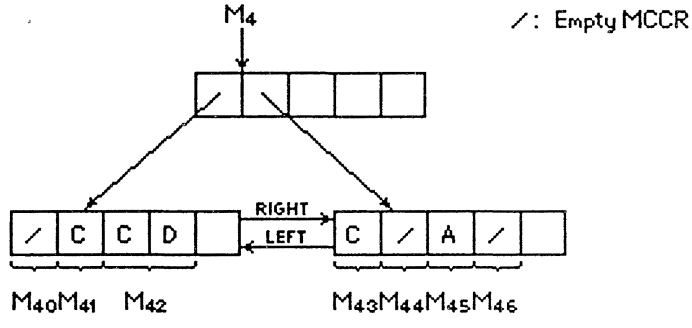
(1) An entry of a node N is corresponding to a segment on the ith axis. The form of an entry is

$$(P, G)$$

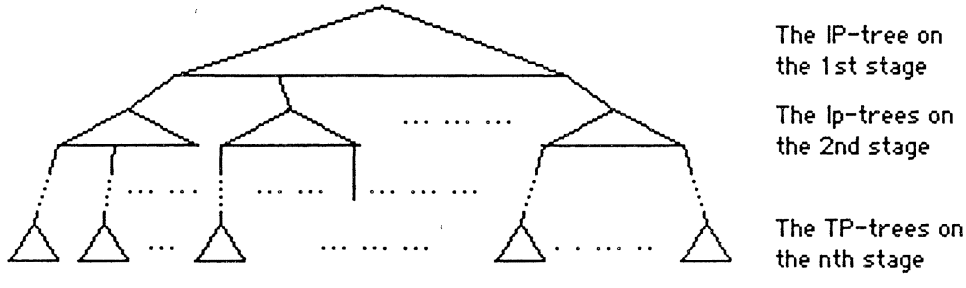
where P is a pointer to a sub-IP-tree if N is a non-leaf node, otherwise, P is pointing to an IP-tree on the (i+1)st stage if $i < n-1$ or to a TP-tree if $i = n-1$. The value of G is the value of the starting point of the segment corresponding to this entry. Namely, in a leaf node, G corresponds to a MCCS; in a non-leaf node G corresponds to the smallest segment covering all segments in the node pointed to



(a)



(b)



(c)

Figure 8. The IP-tree and the TP-tree

(a) The IP-tree corresponding to Figure 7 (a).
(b) The TP-tree corresponding to Figure 7 (b).
(c) The relationship of the IP-tree and the TP-tree.

by P. For example, the values of the entries of the IP-tree in Figure 8(a) are shown in Figure 9(a), if M_i is represented by $[Ml_i, Mu_i)$, for $0 \leq i \leq 7$.

(2) A node of an IP-tree has at most M entries and is stored in one disk page.

(3) A node of an IP-tree has at least m entries unless it is a root, where m = (M+1)/2. If it is a root, it has at least 2 entries unless it is a leaf.

(4) All leaf nodes of an IP-tree appear on the same level.

(5) An entry (P, G) exists in a leaf node of an IP-tree at the ith stage if and only if there is an MCCS which starts at G on the ith axis and is generated by the projection corresponding to the IP-tree. All entries in leaf nodes are sorted in ascending order on the values of Gs.

(6) A leaf node of an IP-tree has two pointers, LEFT and RIGHT. They point to the left and the right neighbor leaf nodes respectively. Leaf nodes of an IP-tree form a doubly linked **Sequential List**.

TP-tree

A TP-tree is an instance of a B^+-tree [Come79][Knut73]. A TP-tree is an instance of the B^+-tree. A TP-tree corresponds to the projection of a sub^{n-1}-space onto the nth axis and indexes the intersection of objects with MCCRs generated from the projection.

A TP-tree has the following properties.

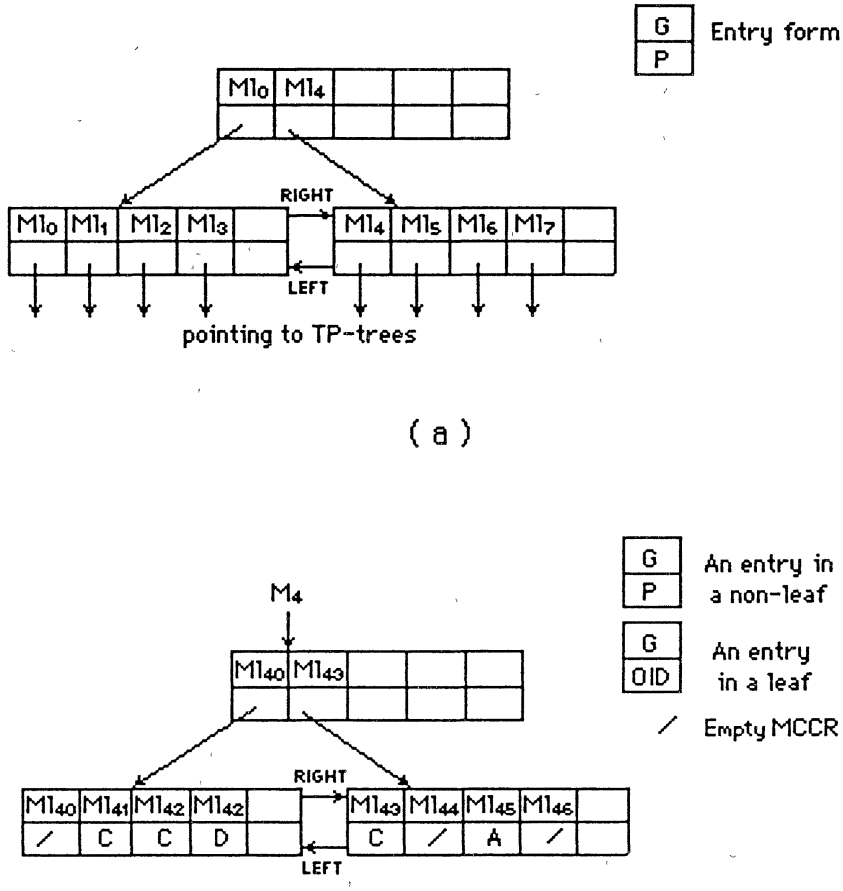


Figure 9. The Values of the Nodes in the IP-tree and the TP-tree

(a) The values of the nodes in the IP-tree corresponding to Figure 8 (a).
(b) The values of the nodes in the TP-tree corresponding to Figure 8 (b).

(1) An entry in a leaf node of a TP-tree represents an MCCR and an object covering the MCCR. The entry is in the form of

(OID, G)

where OID is the identifier of the data tuple of an object covering the MCCR, and the value of G is equal to the lower boundary in the nth dimension of the MCCR (i.e. the starting point of an MCCS on the nth axis). An entry (OID,G) exists in a leaf node of a TP-tree if and only if there is an object with the identifier OID covering an MCCR which is generated by the projection corresponding to the TP-tree and has the lower boundary equal to G in the ith dimension. An empty MCCR has an entry with a dummy OID. For example, the values of the entries of the TP-tree in Figure 8(b) are shown in Figure 9(b), if M_{4k} is represented by $[Ml_{4k}, Mu_{4k})$, for $0 \leq k \leq 6$.

(2) An entry in a non-leaf node of a TP-tree corresponds to a segment on the nth axis and is in the form of

(P, G)

in which the value of G is the smallest value of the Gs in the sub-TP-tree pointed to by P (see Figure 9(b)).

(3) A node of a TP-tree has at most M entries and is stored in one disk page.

(4) A node of a TP-tree has at least m entries unless it is a root, where m = (M+1)/2. If it is a root, it has at least two entries unless it is a leaf.

(5) A leaf node of a TP-tree has two pointers, LEFT and RIGHT. They

point to the left and the right neighbor leaf nodes, respectively. Leaf nodes of a TP-tree form a doubly linked sequential list.

(6) All leaf nodes of a TP-tree are on the same level.

(7) All entries in a leaf node are sorted in a non-descendent order on the values of G.

The entries with the same value of Gs in leaf nodes are stored in consecutive positions in the sequential list and form a **uniform list** corresponding to an MCCR. For example, the entries, C and D, corresponding to M_{42} in Figure 8(b) is a uniform list (also see Figure 9(b)).

Algorithms

Using methods reminiscent to those used for B^+-trees, a topdown search in an IP-tree and a TP-tree is guided by the sorted values of Gs. A horizontal search is along the sequential list through leaves. After an insertion or a deletion, either a split(s) or a merge(s) is(are) applied to some nodes to keep the number of entries within the upper and lower limits and to keep the height of the tree balanced.

Search

The searching algorithm collects all objects which overlap a search region. The algorithm SEARCH in Figure 10 does not change anything in an IDP. As the necessary parameters, ROOT is the address of the root node of an IDP; RECT is the given search region; STAGE

Algorithm SEARCH (ROOT, S_Region, STAGE, n)

Collect all the rectangles overlap the search region from an IP-tree or a TP-tree.

Input: ROOT, the root of an IP-tree or a TP-tree. S_Region, the search region. STAGE, the stage number (dimension number) of the IP-tree or the TP-tree. n, the number of dimensions of the space.

Output: Return RESULT, a set of all the rectangles intersecting the search region.

S1. RESULT <- empty set

S2. Find the rightmost entry, E, among all entries of leaf nodes with value of Gs smaller than RECT.ub$_{STAGE}$.

S3. Scan the sequential list leftward from E. For every entry E_i, let S_i be the segment of E_i and do S4 if INTERSECT(S_Region, S_i) is true.

S4. If STAGE = n then add E_i.OID into RESULT,
else add SEARCH (E_i.P, RECT, STAGE+1, n) into RESULT.

S5. Return (RESULT)

Figure 10. Algorithm SEARCH

is used to discriminate which stage of the IT-tree or the TP-tree being searched, and the value of STAGE is set to be 1 before the whole search; n is the number of dimensions of the space. All these parameters are considered unchanged during search. When SEARCH returns, a set RESULT contains the identifiers of objects each of which corresponds to an object if and only if the object intersects the search region.

Calling the algorithm SEARCH with the value STAGE=1 leads to searching from the root of the IP-tree on the first stage. Since the entries in a node of an IP-tree or a TP-tree are sorted by the values of Gs, then an MCCS identified by a G can be found by a top-down search within an IP-tree or a TP-tree, which is similar to that in a B^+-tree. All MCCSs intersecting the search region are stored consecutively on the sequential list. Recursively searching each tree on the stage STAGE+1 conducts searching at a lower stage. When STAGE = n, a TP-tree is reached. Objects intersecting the search region are stored consecutively in leaf nodes.

Insertion

The algorithm INSERT in Figure 11 inserts an object OBJ into an IDP rooted at ROOT. The parameters STAGE and n have the same meanings as in algorithm SEARCH. OBJ, STAGE and n are unchanged during insertion. ROOT might be changed if a split of node happens to it. In this case, the address of a new root replaces the previous

Algorithm INSERT (ROOT, OBJ, STAGE, n)

Insert an object into an IP-tree or a TP-tree.

Input: ROOT, the root of an IP-tree or a TP-tree. OBJ, the spatial object to be inserted. STAGE, the stage the IP-tree or the TP- tree is on. n, the number of dimensions of the space.

I1. If ROOT is a leaf, then go to I3, else, for every entry E_i in ROOT, let S_i be the segment of E_i and INSERT(ROOT.E_i.P,OBJ, STAGE, n) if INTERSECT(OBJ.RECT, S_i) is true.

I2. Go to I7.

I3. If STAGE = n, go to I6, else for every E_i in ROOT, let M_i be the MCCS of E_i and do I4 if INTERSECT(OBJ.RECT,M_i) is true.

I4. If COVER(OBJ.RECT, M_i), then INSERT (E_i.P, OBJ, STAGE+1, n), else replace the MCCS of E_i by a fully covered MCCS, M', and one or two disjointed MCCSs, duplicate the descendant tree(s) of M_i to attach to every replacing MCCS. Let Ei' be the entry of M', INSERT(Ei'.P,OBJ,STAGE+1,n).

I5. Go to I7.

I6. For every uniform list U_i with an entry in ROOT, let M_i be the MCCR of U_i, if COVER(OBJ.RECT,M_i), then add an entry of OBJ into U_i providing OBJ is not already in U_i, else if INTERSECT(OBJ.RECT, M_i), replace U_i by a fully covered uniform list and one or two disjointed uniform lists. Put the same collection of objects as that of U_i into each of the replacing uniform lists. Add an entry of OBJ into the fully covered uniform list.

I7. If the number of entries in a node is greater than M after insertion, split the node, adjust value of Gs and propagate splits upward if necessary.

Figure 11. Algorithm INSERT

value of ROOT.

Calling algorithm INSERT with the value STAGE=1 inserts an object, OBJ, into the IDP from the root of the IP-tree on the first stage. The object is in the form of (OID, RECT), where OID is the identifier of the data tuple of the object and RECT is the bounding rectangle of the object. The object is inserted into every lower tree or the subtree corresponding a segment S, if INTERSECT(RECT, S) is true. An MCCS, M', in an IP-tree, may not be fully but partially covered by OBJ, that is, INTERSECT(RECT, M') is true but COVER(RECT, M') is false. A partially covered MCCS is replaced by a fully covered MCCS and one or two disjointed MCCSs, which tightly cover the original MCCS. The pointer in the entry of each new MCCS points to a copy of the descendent tree(s) of the original MCCS. Next, the object is inserted into the descendent tree(s) rooted by the fully covered MCCS. When STAGE = n, a TP-tree is reached. The object is inserted into every uniform list whose MCCR is covered fully by OBJ. Similar to the cases of IP-trees, a uniform list covering the upper or the lower boundary of OBJ may not be fully, but partially, covered by OBJ. In this case, the uniform list is replaced by a fully covered uniform list and one or two disjointed uniform lists. Each of the replacing uniform lists has the same collection of objects as the original uniform list. OBJ is put into the fully covered uniform lists.

Deletion

The algorithm DELETE in Figure 12 deletes an object OBJ from an IDP rooted at ROOT. The parameters STAGE and n have the same meanings as in algorithm SEARCH. OBJ, STAGE and n are unchanged during deletion. ROOT might be changed if a mergence of nodes happens to the only two child nodes of ROOT. In this case, the address of a new root which contains the entries in the two children of ROOT, replaces the previous value of ROOT.

Algorithm DELETE proceeds in a manner similar to INSERT but has the opposite function. It deletes an object by removing all entries with OID of the object from the uniform list whose MCCR is covered by the object. After the removal of OBJ, two original MCCSs in an IP-tree (or two original uniform lists in a TP-tree), which are located adjacently and separated by the lower or the upper boundary of the deleted object may have the same collection of objects. In this case, the algorithm removes one of the two MCCSs (or uniform lists) and enlarges the remaining MCCS (or uniform list) to cover the previous two MCCSs (or uniform lists) tightly.

The program for inplementation of Algorithm SEARCH, Algorithm INSERT and Algorithm DELETE is listed in APPENDIX.

Algorithm DELETE (ROOT, OBJ, STAGE, n)

Delete an object from an IP-tree or a TP-tree.

Input: ROOT, the root of an IP-tree or a TP-tree. OBJ, the object to be deleted. STAGE, the stage the IP-tree or the TP-tree is on. n, the number of dimensions of the space.

D1. If ROOT is a leaf, then go to D3, otherwise, for every entry, E_i, of the segment S_i, in ROOT, if COVER(OBJ.RECT, S_i), DELETE(ROOT.E_i.P, OBJ, STAGE, n).

D2 Go to D7.

D3. If STAGE = n, go to D5, else for every entry E_i in ROOT, letM_i be the MCCS of E_i, if COVER(OBJ.RECT, M_i), DELETE(E_I.P, OBJ, STAGE+1, n).

D4. Compare the two MCCSs separated by OBJ.lb_{STAGE} and compare the two separated by OBJ.ub_{STAGE}. If they correspond to the same collection of objects, remove the MCCSs which start at the separating points and remove all their descendant trees. Go to D7.

D5. For every uniform list U_i with an entry in ROOT, let M_I' be the MCCR of U_i and if COVER(OBJ.RECT, M_I'), remove OBJ from U_i.

D6. Compare the two uniform lists separated by OBJ.lb_n and compare the two separated by OBJ.ub_n. If they have the same collection of objects, remove the uniform lists which start at the separating point.

D7. If the number of entries in a node is fewer than m after removing, merge it with its neighbor node, adjust value of Gs and propagate mergences upward if necessary.

Figure 12. Algorithm DELETE

CHAPTER IV

TIME COMPLEXITIES OF IDP

The time complexity of the IDP algorithms is analyzed in two aspects: the time required for page access and the time required for searching within a page.

In the analysis of the performance related to page access, the page access required for an search operation on IDP is discussed. In the analysis of the performance related to time for searching within a page, the number of comparisons for searching an entry within a page is discussed.

Page Access of Search in an IDP

Search by a Point Region

The number of pages accessed is the number of nodes accessed in the IP-trees and TP-trees because every node of an IP-tree or a TP-tree is stored on one disk page. The page access in a search is the function of not only the number and the distribution of objects in a data space but also the size and the position of the search region. In this paper, the analysis of page access proceeds for the cases of point searches only. A point search is a search on a point region. A point region overlaps one MCCS on each stage because MCCSs are

disjoint.

All objects containing the search point are stored in one uniform list in the sequential list of a TP-tree. The pages accessed for a point search are the sum of the heights of trees on every stage in the search path, plus the pages accessed for scanning a uniform list.

Page Access of Search by a Point Region in an IDP

For an IP-tree, the number of total entries of leaf nodes is the number of total MCCSs.

While adding an object into the data space, the bounding rectangle of the object is projected to be segments on each axis. The starting point or the end point of the segment of the projection is either within an existing MCCS and it splits the MCCS into two, or it falls at a boundary of an existing MCCS so it does not create a new MCCS. Thus, inserting an object (projecting an object onto an axis) may increase the number of MCCSs by 0, 1 or 2.

By the definition of MCCS, a projection of an empty space results in 1 MCCS. After loading **N** objects, **u**, the number of MCCSs on an axis, is at most

$$u = 2N+1 \qquad \text{(E1)}$$

2N+1 is the upper boundary of the number of MCCSs.

To calculate the height of an IP-tree or a TP-tree, **f** is denoted to be the fan-out of a non-leaf node in an IP-tree or a TP-tree, and **C** is denoted to be the capacity of a leaf node in an IP-tree or a TP-

tree. Namely, a non-leaf node of an IP-tree or a TP-tree has f entries, a leaf node of an IP-tree or a TP-tree has C entries. In an IP-tree, the number of total entries in leaf nodes is the number of MCCSs on the corresponding axis. If **u** is the number of MCCSs in the leaf nodes of an IP-tree, **Hi**, the height of the IP-tree is

$$Hi = \log_f (u/C)+1 \qquad \text{(E2)}$$

In a TP-tree, the number of total entries in leaf nodes is the sum of all the entries of all uniform lists. If **N'** be the number of all entries in all leaves of a TP-tree, **Ht**, the height of the TP-tree is,

$$Ht = \log_f (N'/C) + 1 \qquad \text{(E3)}$$

A uniform list is the collection of the objects covering a corresponding MCCR. When searching, a uniform list should be scanned sequentially from one end to the other. The page access for scanning a uniform list is the span of the list in pages. If **L** is the length of the uniform list accessed, the number of extra pages accessed in scanning the uniform list is (L-1)/C , where the uniform list is assumed to have equal probability to be stored at any position in the leaf nodes.

The page access for a point search on an IDP is

$$Pa = (\sum_{j=1}^{n-1} Hi'_j) + Ht+(L-1)/C \qquad \text{(E4)}$$

where Hi'_j is the height of the IP-tree on the jth stage accessed in the search path.

A Space with Uniformly Distributed Data

To compare the IDP with the existing structures, the performance of searching the IDP is analyzed in an n-d space which is defined below as a space with uniformly distributed objects.

The data space, R, is represented by an n-d rectangle $R=[sl_1, su_1, sl_2, su_2, \ldots, sl_n, su_n)$, where sl_i and su_i are the lower and the upper boundaries of the rectangle, respectively, in the ith dimension, for $1 \le i \le n$. The data space intersects a certain number, N, of object rectangles with same size $w = w_1 \cdot w_2 \cdot \ldots \cdot w_n$, where w_i is the width of the rectangle along the ith dimension. The distribution of object rectangles is described below.

A super rectangle $Rs = [sl_1-w_1, su_1, sl_2-w_2, su_2, \ldots, sl_n-w_n, su_n)$, which contains the data space R, is latticed with N_i+1 equal intervals each of which is $I_i = (su_i-sl_i+w_i) / (N_i+1)$ along the ith dimension, for $1 \le i \le n$. Then $(N_1+2)\cdot(N_2+2)\cdot \ldots \cdot(N_n+2)$ scaling points, the cross points of lattices, are generated. Each of these points has the coordinates of $(sl_1-w_1+j_1I_1, sl_2-w_2+j_2I_2, \ldots, sl_n-w_n+j_nI_n)$, where $0 \le j_i \le N_i+1$ and $I_i = (su_i-sl_i+w_i) / (N_i+1)$, for $1 \le i \le n$. Then $(N_1+2)\cdot(N_2+2)\cdot \ldots \cdot(N_n+2)$ rectangles of size w are put on the scaling points by overlapping their starting points with these scaling points, one rectangle for each scaling point, where the starting point of a rectangle is defined to be the point with the coordinates of the lower boundaries of the rectangle (Figure 13).

The data space, R, itself now intersects with the certain

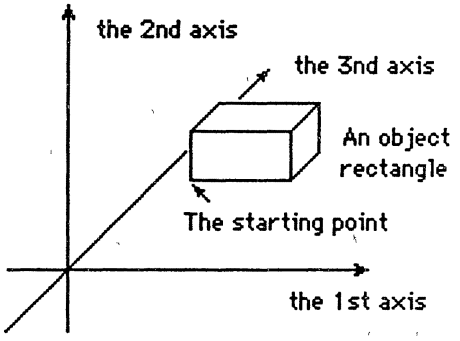


Figure 13. The Starting Point of a 2-Dimensional Rectangle

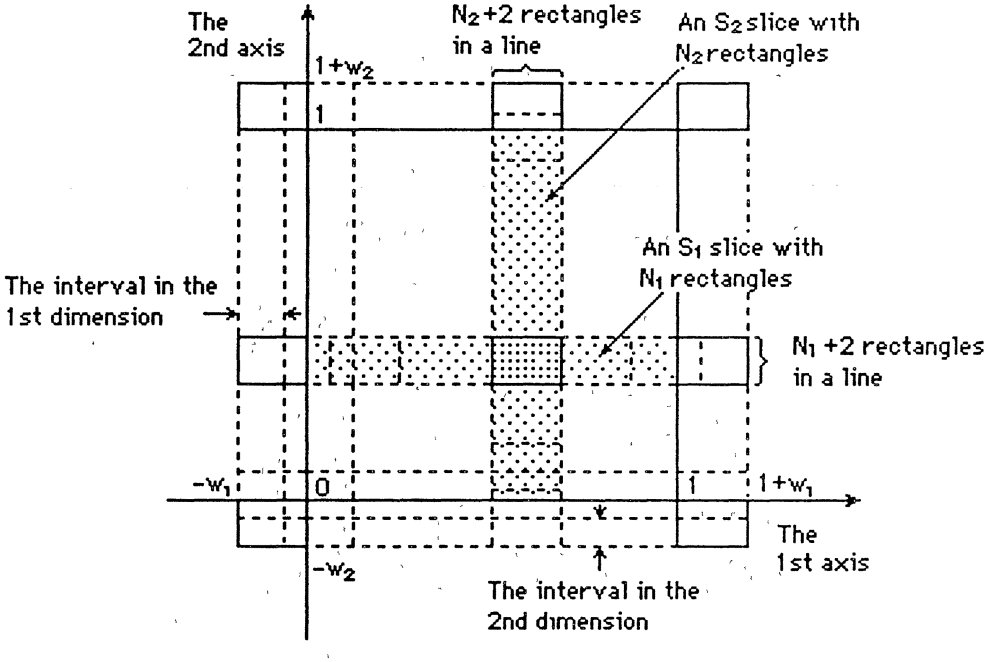


Figure 14. A 2-Dimensional Data Space with Uniformly Distributed Objects

number, $N = N_1 \cdot N_2 \cdot \ldots \cdot N_n$, of object rectangles. Those rectangles which does not intersect with R are considered unrelated to the data space. Those rectangles not included fully in R are cut by the boundaries of R and are considered to be the rectangles of their intersections with R. Every point within R is covered by same number of object rectangles. Figure 14 is an example of a 2-d data space with $sl_1=sl_2=0$ and $su_1=su_2=1$, where object rectangles are uniformly distributed.

Let S_i be the notation of a slice of the space corresponding to the ith dimension, for $1 \leq i \leq n$. An S_i slice is a non-empty n-d rectangle within Rs. A slice contains all the rectangles with their starting points on a line along the ith dimension and has the space of a rectangle which bounds all those rectangles it contains. For the ith dimension, there may be more than one S_i slices. The space occupied by an S_i slice can be represented by the form $[l_{i,1}, u_{i,1}, l_{i,2}, u_{i,2}, \ldots, l_{i,n}, u_{i,n})$, if $k = i$, then $l_{i,k} = sl_i$ and $u_{i,k} = su_i$, otherwise $l_{i,k} = sl_k - w_k + j_k l_k$ and $u_{i,k} = l_{i,k} + w_k$, where $0 \leq j_k \leq N_k$, and $l_k = (su_k - sl_k + w_k)/(N_k+1)$, for $1 \leq k \leq n$ (see Figure 14). The rectangles considered as in an S_i are those totally included in this S_i . The density D_i at a point in an S_i is defined as the number of rectangles which are in this S_i slice and cover this point. Namely, each point in an S_i slice is covered by same number, D_i, of rectangles belonging to this S_i slice, for $1 \leq i \leq n$. A point in the data space, R, is covered by $(D_1 \cdot D_2 \cdot \ldots \cdot D_{i-1} \cdot D_{i+1} \cdot \ldots \cdot Dn)$ S_i slices. Consequently, a point in the data

space R is covered by $D = D_1 \cdot D_2 \cdot \ldots \cdot D_n$ rectangles. D is the density of the data space. For example, a point p in a 2-d data space is covered by D_2 slices of S_1 and each slice of S_1 has D_1 rectangles covering p. So a point p in the data space is covered by $D = D_1 \cdot D_2$ rectangles.

This space with uniformly distributed objects is equivalent to the space of uniform distribution which was introduced by Faloutsos et al [Falo87a][Falo87b]. In a special case of $su_1\text{-}sl_1 = su_2\text{-}sl_2 = \ldots = su_n\text{-}sl_n$, $w_1 = w_2 = \ldots = w_n$ and $N_1 = N_2 = \ldots = N_n$, the number of objects in the data space is $N = N_i^n$, and the density is $D = D_i^n$, for $1 \leq i \leq n$.

Page Access for IDP in a Uniformly Distributed Data Space

In the space with uniformly distributed data, as defined above, the scaling points are generated by latticing the space perpendicularly to the coordinate system. If $N = N_i^n$ and $D = D_i^n$, the number of MCCSs from projecting N rectangles onto the 1st axis or from projecting a sub^i-space onto the (i+1)th axis is equal to that from projecting an S_i slice with the density of D_i onto the ith axis. According to the definition of the space with uniformly distributed objects, this projection places N_i+2 segments on the ith axis by staring them at the positions from $sl_i\text{-}w_i$ to su_i with equal intervals. There are just N_i segments intersect the segment $[sl_i, su_i)$. There are D_i segments containing sl_i are cut by sl_i, and there are D_i segments containing su_i and are cut by su_i , because only the parts of segments within the space are considered. Then, the number of MCCSs in $[sl_i, su_i)$, denoted by u_i, varies with the overlap condition of starting and

end points of these segments. However, there are only two cases.

Case 1: D_i is not an integer so there is no overlap of the starting or end points of segments except the D_i segments which originally started outside of the space are cut to have the same start point at sl_i and the D_i segments which originally ended outside the space are cut to have the same end point at su_i. The number of MCCSs in case 1 is the upper bound of number of MCCS (see Equation E1) minus $2D_i$,

$$u_i = 2N_i+1-2D_i = 2(N_i-D_i)+1 \qquad \text{(E5)}$$

Case 2: D_i is a non-negative integer so every segment starts at the position where another segment ends, except the leftmost D_i segments and the rightmost D_i segments. Among the total $2N_i$ starting and end points, D_i starting points and D_i end points are at the upper and the lower boundaries of the space, respectively. The remaining N_i-D_i starting points and N_i-D_i end points overlap at N_i-D_i different positions between the boundaries and generate N_i-D_i+1 MCCSs, i.e.,

$$u_i = N_i-D_i+1 \quad , \qquad \text{(E6)}$$

Substituting Equations E5 and E6 into Equation E2, the height of an IP-tree on the ith stage is,

$$Hi_i = \log_f ((2(N_i-D_i)+1)/C) +1 \quad , \qquad \text{(E7)}$$

in case 1, and,

$$Hi_i = \log_f ((N_i-D_i+1)/C) +1 \quad , \qquad \text{(E8)}$$

in case 2. Substituting Equation E5 and E6 into Equations E3, the height of an TP-tree is,

$$Ht = \log_f ((2D(N_n-D_n)+1) / C) + 1 \quad , \qquad \text{(E9)}$$

in case 1, and,

$$Ht = \log_f (D(N_n - D_n + 1) / C) + 1 \quad , \qquad \text{(E10)}$$

in case 2. Because the length of a uniform list is the density, D. Equation E4, the total page access for a point search on an IDP is

$$Pa = \left(\sum_{j=1}^{n-1} Hi'_j \right) + Ht + (D-1)/C \qquad \text{(E11)}$$

where Hi'_j is the height of the IP-tree on the jth stage accessed in the search path.

With the same parameters, the Hi_j and Ht calculated by E7 and E9 are greater than by E8 and E10, respectively. So the page access calculated by substituting Equation E7 and Equation E9 into Equation E11 is the upper bound of page access for searching the IDP in a data space with uniformly distributed data objects.

Page Access of Existing Structures

Faloutsos et al [Falo87a][Falo87b] analyzed the page access of R-trees and R+-trees. In a space in which N rectangles are uniformly distributed with an average density of D, where $N = N_l^n$, and $D = D_l^n$. the page accesses in searching on an R-tree and an R+-tree are

$$pr = \log_{fr} (N/Cr) + (1+1/k)^n + (1+1/k/F)^n - 1 \qquad \text{(E12)}$$

$$pr^+ = 1 + \log_{fr} (N/(Cr^{1/n} - D^{1/n})^n) \qquad \text{(E13)}$$

where

pr is the page access for a point search on a optimally organized R-tree,

pr+ is the page access for a point search on a optimally organized R^+-tree,

fr is the fan-out of a non-leaf node in an R-tree or an R^+-tree,

Cr is the capacity of a leaf node in an R-tree or an R^+-tree,

$k = Cr^{1/n} / (D^{1/n}-1)$,

$F = fr^{1/n}$.

pr and pr^+ are the lower boundaries of page accesses in searching because they are for the optimal R-tree and R^+-tree.

Comparison of Page Access with Existing structures

In an n-d space an entry in a non-leaf node of an R- tree or of an R^+-tree contains one pointer and 2n integers to represent an n-d rectangle, an entry in a leaf node of an R-tree or an R^+-tree contains 2n integers to represent an n-d rectangle and one integer for the identifier of an object. An entry in a node of an IP-tree or a TP-tree contains one pointer and one integer for a G, and an entry in a leaf node of an IP-tree or a TP-tree contains two integers for a G and an identifier. A leaf node of an IP-tree or a TP-tree needs two extra pointers for the sequential list.

With the assumption that the size of a pointer, an integer or an identifier is 4 bytes, an entry of an R-tree or an R^+-tree is of size 4(2n+1) bytes, an entry in an IP-tree or a TP-tree is of size 8 bytes. By the assumption that every page needs 4 bytes for the page head, the typical fan-outs and capacities of a 2-d index in the cases of typical page size of 512 bytes and 1024 bytes are in Table 1.

The numbers of pages accessed in a point search in the IDP, the R-tree and the R^+-tree vary with several parameters. In comparisons of page accesses between the IDP and existing structures, Equation E11 in which E7 and E9 are substituted is used for calculating the page access in searching om the IDP, and E12 and E13 are used for the R-tree and the R^+-tree, respectively. In the calculation, one parameter is chosen to be variable and all others fixed. The space is assumed to have $N=N_i^n$ and $D=D_i^n$, $1 \le i \le n$.

From Figure 15 to Figure 20 ,the two cases of page accesses of searching in IDP by a point region are compared with the lower boundaries of R-trees and R^+-trees in point searches of 2 and 3-dimensional spaces with uniformly distributed objects. Page size are chosen to be 512 and 1024 bytes.

In the spaces with low object densities of $1 \le D \le 10$, The performances of searching in IDP are similar to that of R^+-tree and better than R-tree in the case of 1024 byte page (Figure 15 and Figure 16), And even better in the case of 512 byte page (Figure 17 and Figure 18).

In the spaces with high object densities of $10 \le D \le 80$, the performance of IDP in searching is better than that of R-tree and R^+-tree, and even better with the increase of the object density (Figure 19 and Figure 20).

TABLE 1

FAN-OUTS AND CAPACITIES OF A NODE

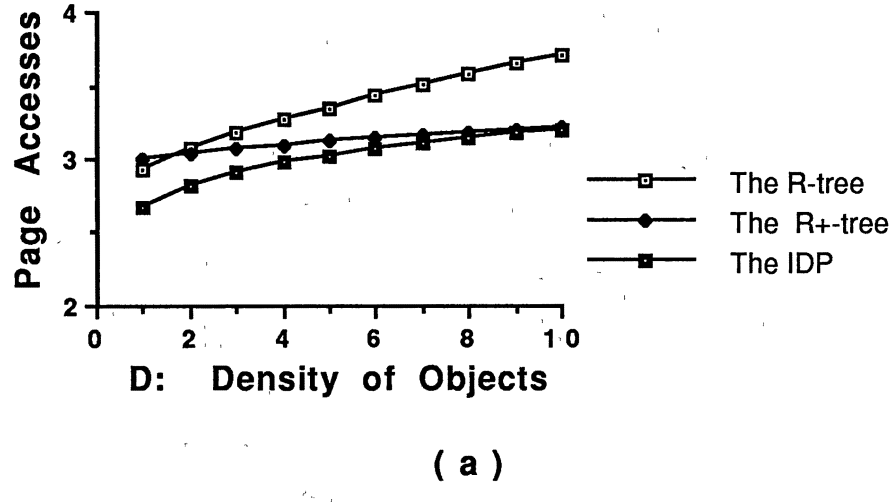
(a)	2-dimensional index		
		Page size = 512 bytes	Page size = 1024 bytes
	f r	(512-4)/20 = 25	(1024-4)/20 = 51
	Cr	(512-4)/20 = 25	(1024-4)/20 = 51
	f	(512-4)/8 = 63	(1024-4)/8 = 127
	C	(512-12)/8 = 62	(1024-12)/8 = 126
(b)	3-dimensional index:		
		Page size = 512 bytes	Page size = 1024 bytes
	f r	(512-4)/20 = 18	(1024-4)/20 = 36
	Cr	(512-4)/20 = 18	(1024-4)/20 = 36
	f	(512-4)/8 = 63	(1024-4)/8 = 127
	C	(512-12)/8 = 62	(1024-12)/8 = 126

fr: Fan-out of R-tree and R^+-tree.

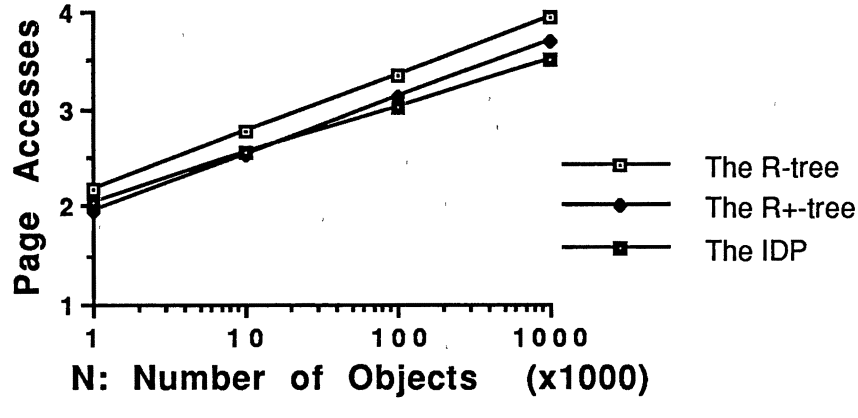
Cr: Capacity of R-tree and R^+-tree.

f: Fan-out of IDP.

C: Capacity of IDP.



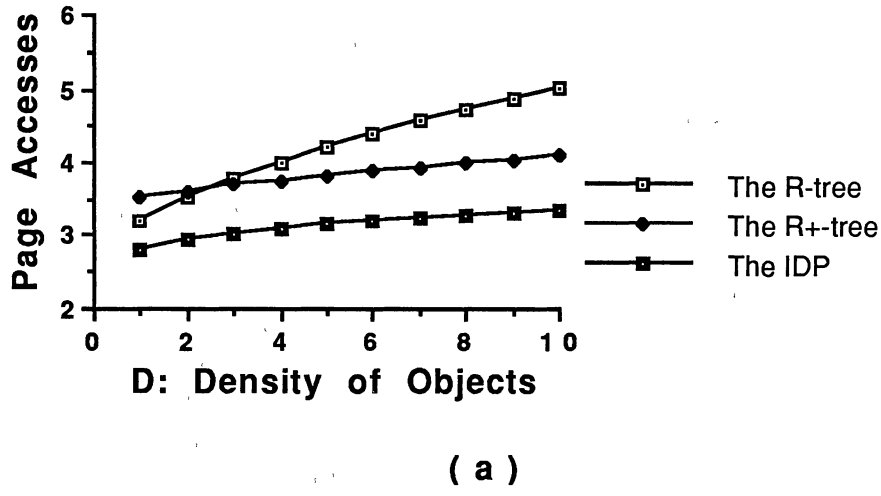
(a)



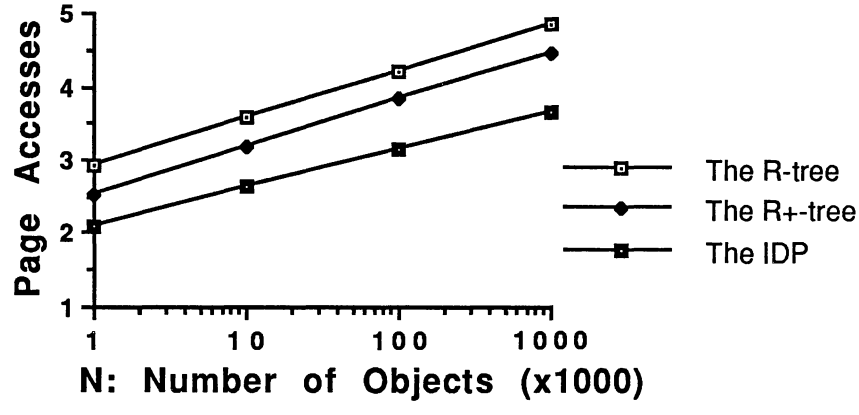
(b)

Figure 15. Page Accesses as Functions of D and N in 2-dimensional Space (Page size = 1024 bytes)

(a) N = 100,000. (b) D = 5.



(a)



(b)

Figure 16. Page Accesses as Functions of D and N in 3-dimensional Space (Page size = 1024 bytes)

(a) N = 100,000. (b) D = 5.

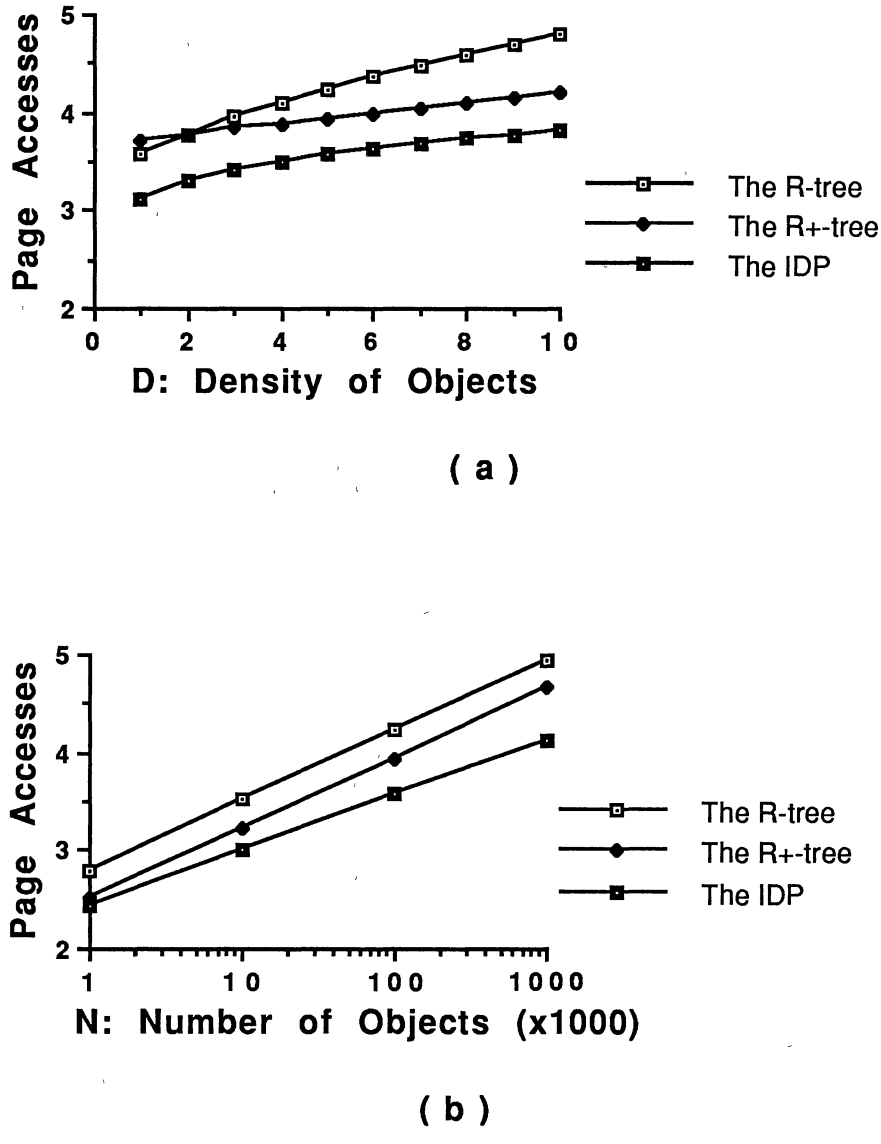
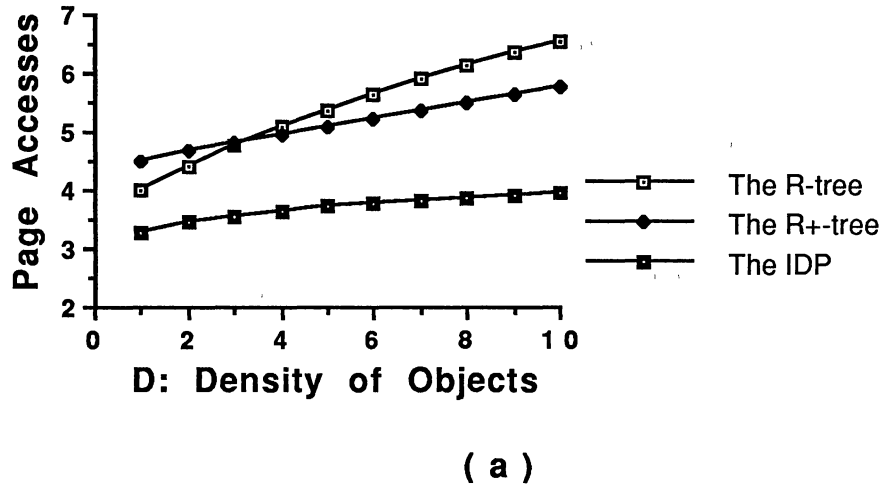
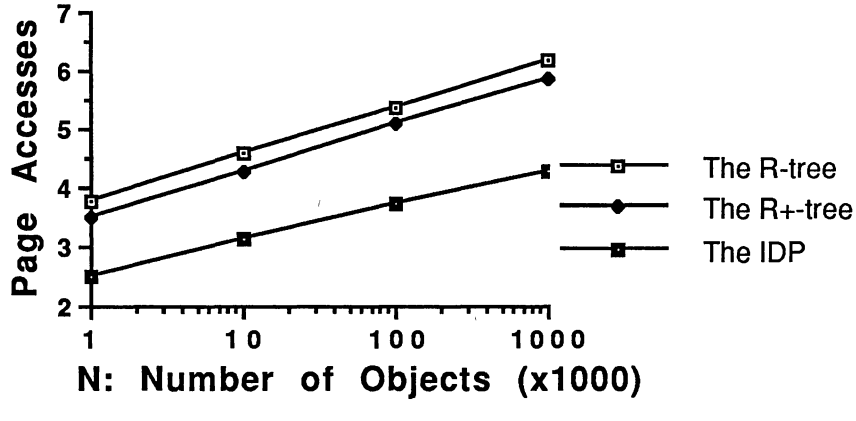


Figure 17. Page Accesses as Functions of D and N in 2-dimensional Space (Page size = 512 bytes)

(a) N = 100,000. (b) D = 5.



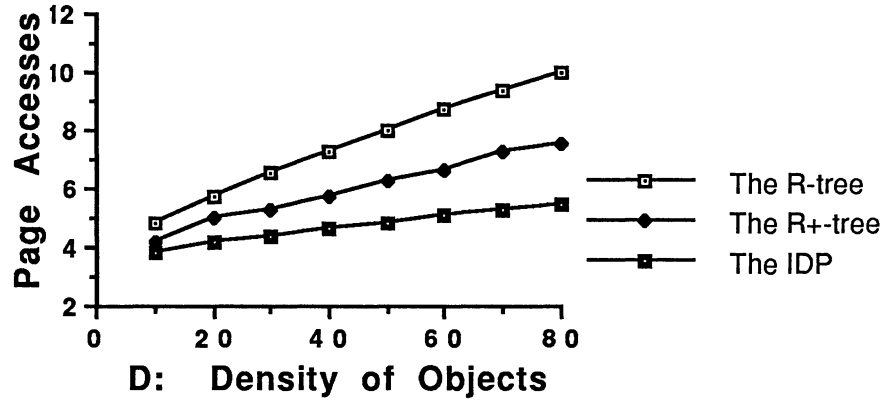
(a)



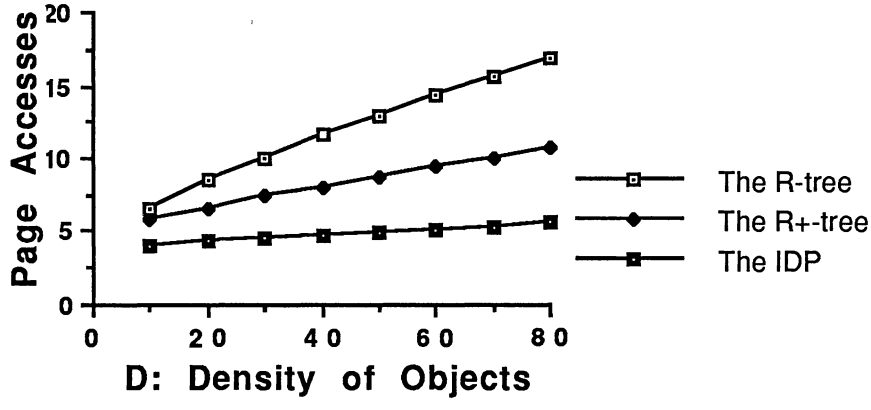
(b)

Figure 18. Page Accesses as Functions of D and N in 3-dimensional Space (Page size = 512 bytes)

(a) N = 100,000. (b) D = 5.



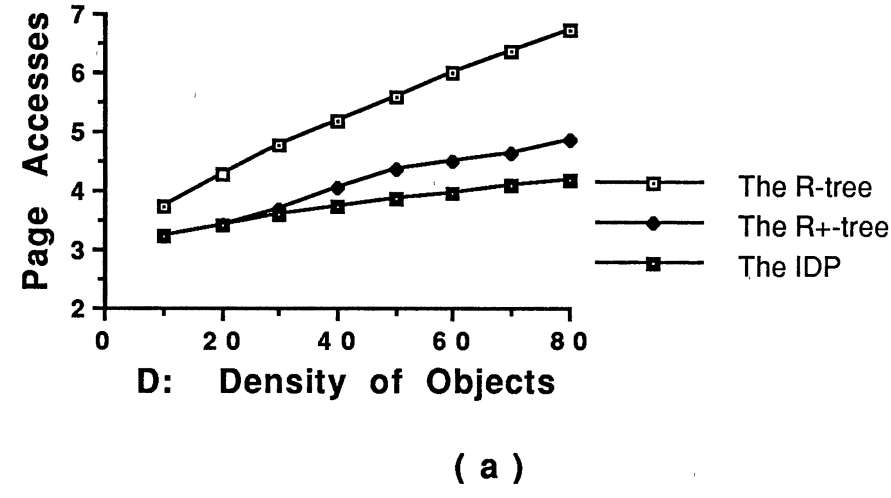
(a)



(b)

Figure 19. Page Accesses as Functions of D
(Page size = 512 bytes,
N = 100,000)

(a) 2-dimensional space,
(b) 3-dimensional space.



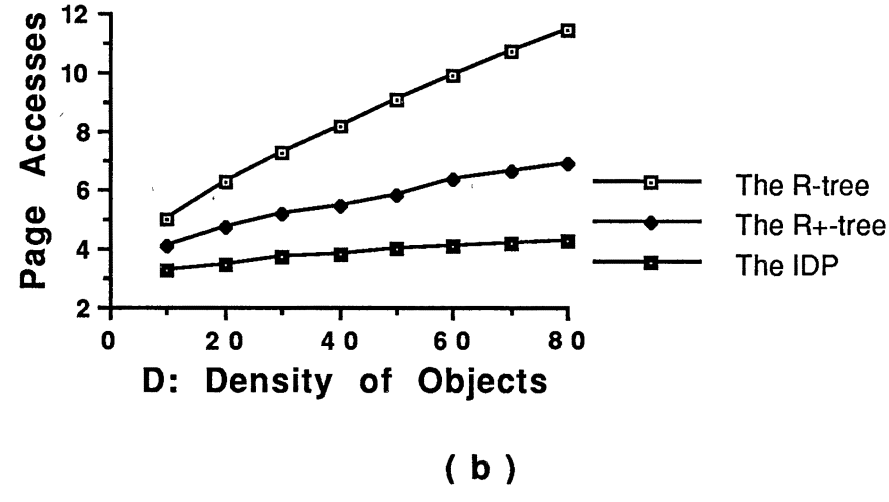


Figure 20. Page Accesses as Functions of D
(Page size = 1024 bytes,
N = 100,000)

(a) 2-dimensional space,
(b) 3-dimensional space.

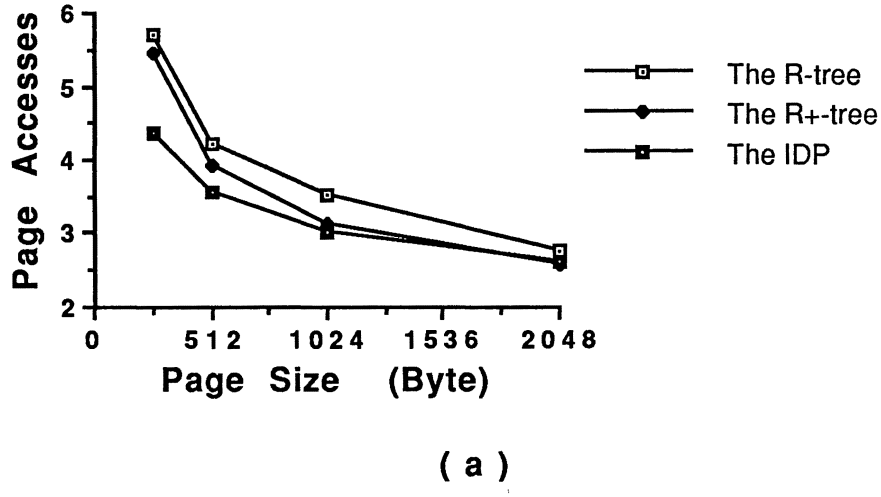
The performance of searching in IDP is not so sensitive to the page size as the R-tree and the R+-tree. Figure 21 shows the page access as the function of page sizes in 2-d and 3-d spaces. The page size varies from 256 bytes to 2048 bytes. The object density, D, is fixed to be 5, and the number of objects, N, is fixed to be 100,000. The performance of searching in the IDP is much better than the R-tree and the R+-tree when the page size is small.

Time for Searching an Entry within a Page

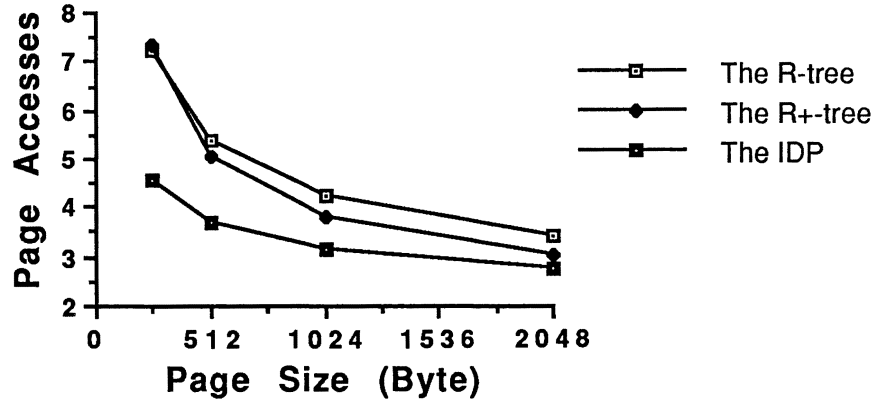
In addition to the time required for a page access in an operation on an index system, the time required by the CPU to process a page of data is another important criterion. Although a page accessing may be more time consuming, the processing of the data in a page may require more CPU time. It is common that CPU time is a more popular performance measure, especially in some multi-user system.

In either an IP-tree or a TP-tree, the entries in a node are sorted on the values of their Gs. Searching an entry among M entries in a node requires $\log_2 M$ comparisons on the average if the binary search method is employed.

The entries in a node of an R-tree or an R+-tree are unsorted. When process a page in search on an R-tree, all entries must be accessed even in a point search because rectangles of entries in a node of an R-tree can overlap each other. In the case of an R+-tree,



(a)



(b)

Figure 21. Page Accesses as Functions of Page Size

D = 5, N = 100,000,
(a) 2-dimensional space,
(b) 3-dimensional space.

a linear search is necessary also. For a search by a non-point region, every entry in a node of an R^+-tree should be checked to determine whether the entry overlaps the search region. For a point search, only one entry in a node of an R^+-tree can overlap the point search region. The searching among the entries in a node of an R^+-tree stops when an entry containing the point search region is found. But the point search region might be in a position which is not covered by any entry in a node. In this case, every entry should be checked. When a linear search is applied and every entry in a node has the same possibility to contain the point search region, if the possibility of the point search region not being covered by any entry is zero, the average number of entries to be checked in a page is 0.5M. otherwise, the average number of entries to be checked is greater than 0.5M. In the worst case, the number of entries to be checked is M, just the same as that of the R-tree.

To determine whether an entry of R-trees or R^+-trees overlap with the search region requires the comparison of upper and lower boundaries in every dimension of the n-d space. The CPU time required is 2nM for processing a page when search in a R-tree or non-zero (or non-unit) region search in an R^+-tree, and is equal to or greater than nM in average when a point search in an R^+-tree, where M is the number of entries in the node being searched, n is the number of dimensions of the space.

If a node of an IDP, an R-tree or an R^+-tree is stored on a disk

page of size 512 bytes or 1024 bytes, the number of the comparisons for processing a page in a point search are in Table 2, in which the numbers of the entries in the nodes of an IDP, an R-tree and an R^+-tree are the fan-outs of these nodes are from Table 1. In the worst case, the IDP and the R-tree retain their numbers of comparisons shown in Table 2, but the numbers of comparisons of the R^+-tree become as large as those of the R-tree.

TABLE 2

NUMBER OF COMPARISONS IN SEARCHING AN ENTRY WITHIN A PAGE

	IDP	R-tee	R^+-tree
2-dimensional index page size = 512 bytes	$\leq \log_2 63 < 6$	$2 \cdot 2 \cdot 25 = 100$	$\geq 2 \cdot 25 = 50$
2-dimensional index page size = 1024 bytes	$\leq \log_2 127 < 7$	$2 \cdot 2 \cdot 51 = 204$	$\geq 2 \cdot 51 = 102$
3-dimensional index page size = 512 bytes	$\leq \log_2 63 < 6$	$2 \cdot 3 \cdot 18 = 108$	$\geq 3 \cdot 18 = 54$
3-dimensional index page size = 1024 bytes	$\leq \log_2 127 < 7$	$2 \cdot 3 \cdot 36 = 216$	$\geq 3 \cdot 36 = 108$

CHAPTER V

OPTIMIZATION OF ORGANIZATION

In the above comparison of page accesses, the IDP is assumed to be fully loaded and the R-tree and the R^+-tree are supposed to be fully loaded as well as organized optimally. With the assumption of full load, the utilization of a memory page (the slots for entries) is 100%. Full utilization of memory decreases the heights of trees. Besides the full utilization of memory, an optimally organized R-tree must have at least the minimum coverage and overlap of nodes, and an optimally organized R^+-tree must have at least the minimum coverage of nodes and the minimum splits of objects [Rous85][Sell87].

Partial Load of Pages

Practically, in dynamic circumstances, memory utilization might be lower than 100% after insertion or deletion operations because of an entry removal or a node split.

A node in an IP-tree, a TP-tree or an R-tree always has a memory utilization between m/M and 1. As instances of B-trees or B^+-trees, these trees have the average memory utilization of at least 70% [Knut73].

An R^+-tree does not have the lower and the upper limit of

entries in a node. The utilization is lower than 70%. Sellis, et al claimed that the memory utilization of the R^+-tree is about 60% which is like that of a k-d-B-tree [Falo87a][Falo87b][Robi81][Sell87]. Because partial load of pages increases the height of IDPs, R-trees and R^+-trees, more pages might be accessed in searching algorithms. In the cases of partial load, the performance of page access in searching in an IDP is not so sensitive as an R-tree or an R^+-tree. Figure 22 shows the page accesses as functions of memory utilization. With the memory utilization less than 100%, the IDP has smaller degradation of search performance with memory utilization decrease than those of the R-tree or the R^+-tree.

The algorithm COMPRESS in Figure 23 optimizes an IP-tree or a TP-tree to be fully loaded.

Organizational Optimization

Because all entries in a node of an IP-tree or a TP-tree are ordered, optimizing IP-trees and TP-trees to fully loaded requires only compressions of the nodes. A compressed IDP have the performance of searching as discussed in Chapter IV.

To compress an IP-tree or a TP-tree need no extra sorting. All the processing is along the scanning of entries on the same level. This is a linear complexity in every level. A parent level has 1/C or 1/f entries of its child level. So the total number of entry moving is

a the sum of a geometric series with a ratio less than 1 and is linear complexity. Obviously every page is accessed only once during compress. The page access for compression is linear too. An insertion or a deletion of an object only changes some of the IP-trees and the TP-trees in an IDP. The optimization required after an insertion or a deletion may be only a local process(es) and can be performed during that insertion or deletion.

In order to maintain the performances of R-trees and R^+-trees in searching, some methods to optimize R-trees and R^+-trees are necessary in the insertion and/or deletion algorithms, or applied periodically. To optimize R-trees and R^+-trees to have the performances in searching, which is discussed in Chapter IV, requires a combinational algorithm. Because,
the criteria for optimization of an R-tree include at least,

(a) minimal total coverage of organized rectangles,

(b) minimal overlap among rectangles on the same level,

(c) maximal memory utilization,

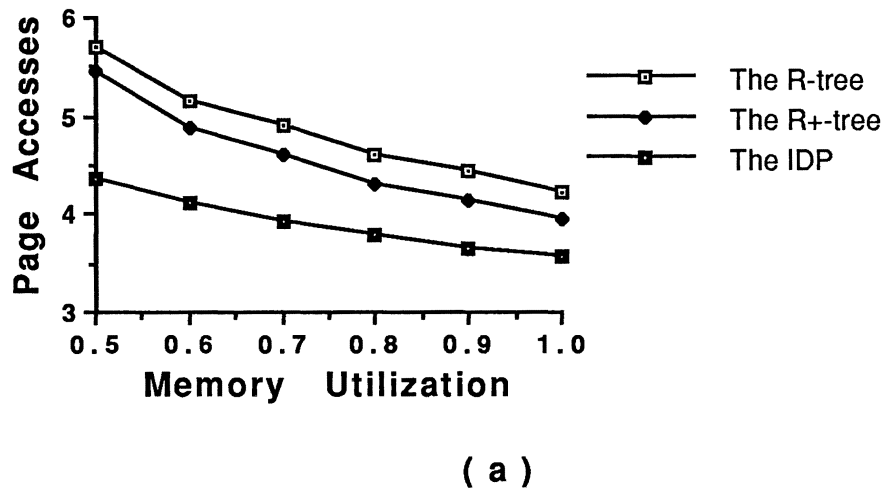
and, the criteria for optimization of an R^+-tree include at least

(a) minimal total coverage of organized rectangles,

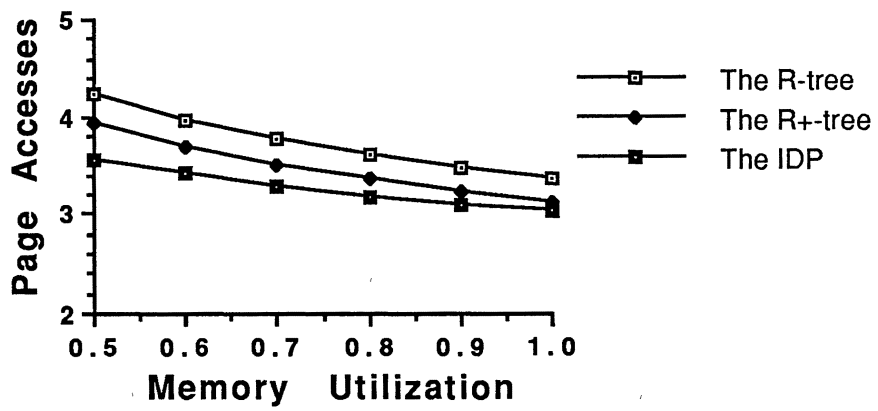
(b) minimal number of rectangle splits,

(c) maximal memory utilization [Rous85][Sell87].

To fulfill all the criteria is obviously an NP problem. Even fulfilling one of the criteria (a) and (b) of the R-tree or (a) and (b) of the R^+-tree requires combinational operations. Fulfilling only one of the criteria cannot improve an R-tree or an R^+-tree to have the



(a)



(b)

Figure 22. Page Accesses as Functions of Memory Utilization

2-dimensional space, D = 5,
N = 100,000, (a) 512B page,
(b) 1024B page.

Algorithm COMPRESS (ROOT)

Compress an IP-tree or a TP-tree to make the memory utilization as high as possible.

Input: ROOT, the root of an IP-tree or a TP-tree.

Output: A fully loaded IP-tree or TP-tree rooted at ROOT.

C1: Guided by the value of Gs in the nodes of the tree to find the leftmost entry in the leftmost node. Let L point to the node.

C2: If L = ROOT, return.

C3: Free all the storage of intermediate nodes. Scan the sequential list from L. Adjust all entries to the left to make all nodes on the leaf level fully loaded except the rightmost two. These two can have fewer than M and at least m entries. Free the storage of empty nodes. Let S be the list of nodes on the leaf level.

C4: While S level has more than one nodes, do C5.

C5: Make a list of nodes to be the parent level of S level. Fully load each of the nodes in the parent level except the rightmost two. These two can have fewer than M entries but at least m entries. Let S be the list of the nodes on the parent level.

C6: Let ROOT be the address of the only node on S level. return.

Figure 23. Algorithm COMPRESS

performances discussed in Chapter IV.

Instead of an NP solution, Roussopoulos and Leifker [Rous85] introduced a PACK algorithm to reorganize a naturally grown R-tree. The comparison of page access required by the packed R-tree with the natural R-tree from Guttman's algorithm [Gutt84] by the experimental results over 100 random searches is shown in Table 3.

Table 3 shows that optimization is very important in the use of R-trees. The PACK algorithm is transformed as a routine in insertion algorithm but is not in the deletion algorithm of the R^+-tree [Sell87]. So periodical optimization is important for R^+-trees if deletion happens frequently.

PACK is a non-NP algorithm but it only results in sub-optimally organized R-trees or R^+-trees because the pack algorithm only pursue the criterion of a near minimal coverage of organized rectangles. Even as a near fulfilling of one of the criteria, PACK algorithm requires the sort of all rectangles by their lower boundaries an upper boundaries for every level of nodes along every dimension. Only for the sort of rectangles on the leave level, at least $n \cdot N \log_2 N$ comparisons are needed, where n is the number of the dimensions of the space, N is the number of the data objects. Page access during a sort is unpredictable because the values being compared can be on different pages. The cost of PACK is so high that the R-tree and the R^+-tree are suitable only in semi-dynamic circumstances.

TABLE 3

COMPARISON BETWEEN NATURAL AND PACKED R-TREES
ON PAGE ACCESS IN A SEARCH

The number of data objects	Page access of R-tree from Guttman's INSERT	Page access of R-tree from PACK algorithm
10	2.217	1.424
50	7.775	2.282
100	12.955	3.645
200	17.870	3.873
300	20.838	5.397
400	28.953	5.418
500	36.132	5.466
600	70.799	5.276
700	45.924	5.604
800	55.462	5.730
900	63.595	6.071

CHAPTER VI

SUMMARY AND CONCLUSIONS

Query for spatial data objects by region is required by pictorial databases and spatial data retrieve systems. Query for spatial data objects by region is different from the queries for traditional types of data and can not be supported by traditional indexes. Without indexes, a query for spatial data objects is processed very slowly.

In order to handle spatial data efficiently, as required in computer-aided design and geometric data applications, a database system needs an index mechanism to retrieve data items quickly according to their spatial locations. However, traditional indexing methods are not well suited to data objects with non-zero size located in multidimensional space.

Two existing index structures, the R-tree [Gutt84] and the R^+-tree [Sell87], were introduced to meet these needs. Both R-trees and R^+-trees are extensions of B-trees which maintain their balanced heights as well as logarithmic page access.

However, either the R-tree or the R^+-tree has some disadvantages. A new index structure, the IDP, which is proposed in this thesis, has the following advantages:

(1) The performance of page access in searching in an IDP is of logarithmic to the number of data objects. In the spaces where data

objects are uniformly distributed, the page access in a point search in an IDP is better than that in an R-tree and similar to or better than that in an R^+-tree, and even better when page size is small.

(2) The algorithmic complexity for searching for an entry in a node of an IDP is logarithmic to the number of entries in the node. It is less than it is in a node of either an R-tree or an R^+-tree.

(3) The algorithm of organizing an optimal IDP has a time complexity lower than that of the algorithm of organizing either an optimal R-tree or an optimal R^+-tree.

The suggested future work includes:

(1) Analysis of time complexities for search in the IDP in more general cases and for insertion and deletion in the IDP.

(2) The implementation of IDP to the indexes of practical data collections and the experimental analysis of the performance of IDP in retrieval of data objects with different distributions in sizes and positions.

REFERENCES

[Bent75] Bentley, J.L., Multidimensional Binary Search Trees Used for Associative Searching, *CACM*, 18,9(Sept. 1975), 509-517.

[Bent79] Bentley, J.L., Friedman, J.H. Data Structures for Range Search. *Computing Surveys*, 11.4(1979), 397-409.

[Chan81a] Chang, S.K., Pictorial Information System: Guest Editor's Introduction, *IEEE Computer*, 14, 11(Nov. 1981), 10-11.

[Chan81b] Chang, S.K., Kunii, T.L., Pictorial Data-Base System, *IEEE Computer*, 14, 11(Nov. 1981), 13-21.

[Chan81c] Chang, N.S., Fu, K.S., Picture Query Languages for Pictorial Data-base Systems, *IEEE Computer*, 14,11(Nov. 1981).

[Come79] Comer, D., The Ubiquitous B-tree, *Computing Surveys*, 11, 2(1979), 121-138.

[Falo87a] Falousos, C., Sellis, T., Roussopoulos, N., Analysis of Object Oriented Spatial Access Methods, *Proc. of the ACM SIGMOD*, 1987, 426-439.

[Falo87b] Faloutsos, C., Sellis, T., Roussopoulos, N., Metaxas, D., Object Oriented Access Methods for Spatial Objects: Analysis for Multi-Dimensional Spaces, UMIACS-TR-87-54 CS-TR-1940 Univ. of Maryland, Oct. 1987.

[Free89] Freeston, M. W., A Well-Behaved File Structure for the Storage of Spatial Objects, *The First Symposium on the Design*

and Implementation of Large Spatial Database, Santa Borara, CA., July 1989.

[Good89] Goodman, A.M., Hralick, R.M., Shapiro, L.G., Knowledge-Based Computer Vision, *IEEE Computer*, 22.12(1989), 43-54.

[Gros89] Grosky, W.I., Mehrotra, R., Guests' Introduction: Image Databases Management, *IEEE Computer*, 22.12(1989), 7-8.

[Gunt89] Gunther, O., The Design of the Cell tree: An Object-Oriented Index Structure for Geometric Databases, *Proc. of the 5th International Conf. on Data Engineering*, Los Angeles, Feb. 6-10, 1989.

[Gutt84] Guttman, A., R-trees: A Dynamic Index Structure for Spatial Searching, *Proc. of the ACM SIGMOD*, 1984 June, 47-57.

[Knut73] Knuth, D.E., *The Art of Computer Programming*, Vol.3. Addison-Wesley Publishing, 1973.

[Robi81] Robinson, J.T., The k-d-B-tree: A Search Structure for Large Multidimensional Dynamic Indexes, *Proc. of the ACM SIGMOD*, 1981, 10-18.

[Rous85] Roussopoulos, N., Leifker, D., Direct Spatial Search on Pictorial, Databases Using Packed R-trees. *Proc. of the ACM SIGMOD*, 14.4(1985), 17-31.

[Rous88] Roussopoulos, N., Faloutsos, C., Sellis, T., An Efficient Pictorial Database System for PSQL, *IEEE Transactions on Software Engineering*, 14.5(1988), 639-650.

[Same84] Samet, H., Quadtrees and Related Hierarchical Data Structures, *Computing Surveys*, 16.2(1984), 187-260.

[Same90] Samet, H., *The design and Analysis of Spatial Data Structures*, Addison-Wesley Publishing, 1990.

[Sell87] Sellis, T., Roussopoulos, N., Faloutsos, C., The R+-tree: A Dynamic Index for Multidimensional Objects, *The 13th International Conf. on VLDB*, Brighton, 1987, 507-518.

APPENDIX

PROGRAM FOR IMPLEMENTATION OF ALGORITHMS

```
/*============================================================

            Program name:  IDP_ALGORITHMS
            Language: C

 This program performs the search, insertion and deletion on an
Index by Dimensional Projection (IDP).
      1. search an region:
          search(region, root, 1)
          region is the search region of the type 'rectangle'.
          root is the soot of the whole IDP and is the type of
            '*page'.
         This function returns a  list of identifiers of objects.
          An object is in the list if and only if it overlaps
          the given region.
      2. insert an object into an IDP:
          insert(pointer_of_root, object,1)
          pointer_of_root is the pointer of the root of the
            whople IDP.  This pointer is of the type '**page'.
          object is the object to be inserted. The type is
            'object'.
      3.  delete an object from an IDP:
           delete(root, object, 1)
           root is the root of the whole IDP.  The type of root
             is  '*page'.
           object is the object to be deleted and is of the
             type  'object'.
          This function returns the root of the whole IDP after
           the removal of the given object.
 The parameters of the IDP on which the operations are applied
are set in the '#define' lines at the beginning of the program.
 This program contains three auxiliary functions to output
the contents of an IDP or the result of an operation:
      seqlst, output a sequential list of a TP-tree.
      tplst, output the sequential lists of descendent trees.
      prtolst, output a list of object from a search.

============================================================*/


#include <stdio.h>

#define dim 2     /* dimension */
#define capa 5    /* capasity of a leaf */
#define fanout 5 /* fanout of a non-leaf */
#define bound   10000/* bound of the space */

#define IE pb.il.entry
#define NE pb.nl.entry
#define TE pb.tl.entry
```

```
struct st_nlentry /* entry of a non-leaf in IP */ {
   int g;     /* guide post */
   struct st_page *p;    /* pointer to a child */
  };

struct st_tlentry  /* entry of a leaf in TP */ {
   int g;     /* guide post */
   int tplid; /* tuple id */
  };

struct nleaf /* non-leaf node */ {
   struct st_nlentry entry[fanout];
  };

struct tleaf /* leaf in TP */ {
   struct st_tlentry entry[capa];
   struct st_page *left, *right;
  };

struct ileaf /* leaf in IP */ {
   struct st_nlentry entry[capa];
   struct st_page *left, *right;
  };

typedef struct st_page /* a page */ {
   char flag;
   int load;
   union {
       struct tleaf tl;
       struct ileaf il;
       struct nleaf nl;
     } pb;
  } page;

typedef struct st_rectangle    /* n-dimensional rectangle */ {
   int lb[dim], ub[dim];       /* lower and upper bounds */
  } rectangle;

typedef struct st_obj   /* spatial object */ {
   rectangle rect; /* enclosing rectangle */
   int tplid;        /* tuple id */
  } object;

typedef struct objlist   /* node for the linked list of objects */ {
 int tplid;          /* tuple id of the object */
 struct objlist *next;   /* the next node */
 }  objlist;

/* functions in this program */

page *splitm();
void addentry();
void putm();
```

```
int rightmost();
page *tpinsert();
void dupulst();
page *rebuild();
void rshift();
void dupent();
void leftentry();
void rightentry();
void addobj();
page *duplicate();
void insert();
void newroot();
objlist *search();
void prtolst();
void accobject();
page *del();
page *delete();
void deln();
void deli();
page *delt();
void rmvulst();
int sameobj();
int samelst();
void rmventry();
void mvleft();
void freesub();
objlist *collulst();
void rearrange();


FILE *fp, *ofile;

/*===== insert inserts an object into a TP-tree or an IP-tree */

void insert(root, obj, stage)   page **root;     /* root */
 object obj;        /* object */
 int stage;  /* the stage root is at */

 {
  page *brother;

  if(*root==0)   /* empty tree */ {
      *root=(page*)calloc(1,sizeof(page));
      (*root)->load=2;
      if(stage<dim)     /* IP */ {
          (*root)->flag='i';
          (*root)->IE[0].g=0;
          (*root)->IE[0].p=0;
          (*root)->IE[1].g=bound;
          (*root)->IE[1].p=0;
        }
       else  /* TP */ {
          (*root)->flag='t';
```

```
        (*root)->TE[0].g=0;
        (*root)->TE[0].tplid=0;
        (*root)->TE[1].g=bound;
        (*root)->TE[1].tplid=0;
       }
    }
  if(stage<dim) {
      if((brother=splitm(*root,obj,stage,1)) != 0)
        newroot(root,brother);  /* split the MCCS covers the upper bound */
      if((brother=splitm(*root,obj,stage,0)) != 0)
        newroot(root,brother);  /* split the MCCS covers the lower bound */
    }
   else
      (*root)=tpinsert(*root,obj);  /* insert into a TP tree */
 }


/*===== splitm splits an MCCS. If the MCCS is contains the lower bound
of the object, subsidiary operations to insert object into all covered
MCCS are invoked */

page *splitm(root, obj, stage, upper)
 page *root;  /* root */
 object obj;  /* object */
 int stage,upper; /* upper=1 means the split is by the upper bound */

 {
  char f;
  int right, cutpoint, high, left, left1, rtneib, i, m;
  page *brother, *tree, *newson, *newtree, *leftnode,
       *node1, *rtnode, *head;

  if(upper==1)
    cutpoint=obj.rect.ub[stage-1];
   else
    cutpoint=obj.rect.lb[stage-1];
  f=root->flag;

  if(stage<dim) /* IP */ {
       right=rightmost(root, cutpoint);
       left=right;  leftnode=root;
       if(f=='i')  /*  leaf */ {
          rightentry(root,right,&rtnode,&rtneib);
          if(rtnode->IE[rtneib].g==cutpoint)   {      /* not partially covered */
              brother=0;
            }
           else     {
              tree=duplicate(root->IE[right].p,&head);
              if(root->load<capa)   {  /* has a free room */
                                 /* paritally covered */
                 putm(root,right,cutpoint,tree);
                        /* duplicate lower structures */
                 brother=0;
```

```
        }
      else      {     /* no free room */
        brother=(page *)calloc(1,sizeof(page));
        high=(right>=capa/2)? 1: 0;
        m=(capa+high)/2;
        for(i=0;i<capa-m;i++)   /* move half node to
                                   the new node */
          brother->IE[i]=root->IE[m+i];
         brother->flag='i';
        brother->load=capa-m;
        root->load=m;
        brother->pb.il.right=root->pb.il.right;
        brother->pb.il.left=root;
        if(root->pb.il.right!=0)
          root->pb.il.right->pb.il.left=brother;
        root->pb.il.right=brother;
        if(high==1) {     /* cutpoint is in th enew node */
           putm(brother,right-m,cutpoint,tree);  /* add a MCCS */
           left=right-m; leftnode=brother;
          }
         else {      /* cutpoint is in the left part */
          putm(root,right,cutpoint,tree); /* add a MCCS */
          }
      }
    }
  if(upper==0) {    /* MCCS is concerned with the lower bound */
      rightentry(leftnode,left,&leftnode,&left);
      while(leftnode!=0 && leftnode->IE[left].g
            < obj.rect.ub[stage-1])   {
              /* insert obj into every covered MCCS */
        if(leftnode!=0)
          if(stage+1<dim)        /* into IP */
            insert(&(leftnode->IE[left].p),obj,stage);
           else                  /* into TP */
            leftnode->IE[left].p
              =tpinsert(leftnode->IE[left].p,obj);
          rightentry(leftnode,left,&leftnode,&left);
       }
    }
 } /* f=='i' */
else { /* f=='n', non-leaf */
  if((newson=splitm(root->NE[right].p,obj,stage,upper))
       != 0) {          /* lower node splitted */
      if(root->load<fanout) {
        addentry(root,right,newson);
        brother=0;
       }
      else {
        brother=(page *)calloc(1,sizeof(page));
        high=(right>=fanout/2)? 1: 0;
        m=(fanout+high)/2;
        for(i=0;i<=fanout-m;i++)
          brother->NE[i]=root->NE[m+i];
```

```
                brother->flag='n';
                brother->load=fanout-m;
                root->load=m;
                if(high==1) /* add to the right part */
                   addentry(brother,right-m,newson);
                 else          /* add to the left part */
                   addentry(root,right,newson);
              }
            }
          else     /* no split */
            brother=0;
        }
       return(brother);
     } /* stage < dim */
   else  /* stage=dim, insert to a TP tree */
    return(tpinsert(root,obj));
 }


/*===== newroot makes 'oldroot' be the father of the original
'oldroot' and 'brother' */

void newroot(oldroot,brother)
 page **oldroot, *brother;

 {
  page *nroot;

  nroot=(page *)calloc(1,sizeof(page));
  nroot->load=2;
   nroot->flag='n';
  nroot->NE[0].p=(*oldroot);
  nroot->NE[1].p=brother;
  if((*oldroot)->flag=='n')   {
     nroot->NE[0].g=(*oldroot)->NE[0].g;
     nroot->NE[1].g=brother->NE[0].g;
    }
   else    {
     nroot->NE[0].g=(*oldroot)->IE[0].g;
     nroot->NE[1].g=brother->IE[0].g;
    }
  *oldroot=nroot;
 }


/*===== addentry adds an entry ( a son, 'newson') to a node */

void addentry(node, pos, newson)
 page *node, *newson;       /* node and the entry to be added in */
 int pos;    /* the position at the left of the new entry */
 {
  int i;
```

```
   for(i=node->load-1;i>pos;i--)
    node->NE[i+1]=node->NE[i];
   if(newson->flag=='n')
    node->NE[pos+1].g=newson->NE[0].g;
   else
    node->NE[pos+1].g=newson->IE[0].g;
   node->NE[pos+1].p=newson;
   node->load++;
  }

/*===== putm adds an MCCS into a leaf node */

void putm(node, pos, startpoint, tree)
 page *node, *tree;    /* the node, the tree pointed by the new MCCS */
 int pos, startpoint;  /* the position at the left of the new MCCS,
                          the guide post of the MCCS */


  {
   int i;

   for(i=node->load-1;i>pos;i--)
    node->IE[i+1]=node->IE[i];
   node->IE[pos+1].g=startpoint;
   node->IE[pos+1].p=tree;
   node->load++;
  }

/*===== rightmost returns the position of the greatest guidepost less than
'cutpoint' in 'node' */

int rightmost(node, cutpoint)
 page *node;
 int cutpoint;

  {
   int l, h, mid, g;
   char f;

   f=node->flag;
   l=0; h=node->load-1;
   while(l!=h)    {
      mid=(l+h+1)/2;
      switch (f)  {
         case 'n':  g=node->NE[mid].g; break;
         case 'i':  g=node->IE[mid].g; break;
         case 't':  g=node->TE[mid].g; break;
        }
      if(g>=cutpoint)
        h=mid-1;
       else
        l=mid;
```

```
    }
  return(l);
 }


/*===== tpinsert inserts an object, 'obj', into a TP-tree */

page *tpinsert(root, obj)
 page *root;
 object obj;

 {
  page *r, *r1;
  int split, right, right1;

  if(root==0)   {
     r=(page*)calloc(1,sizeof(page));
     r->load=2; r->flag='t';
     r->TE[0].tplid=r->TE[1].tplid=0;
     r->TE[0].g=0;
     r->TE[1].g=bound;
     root=r;
   }
  else
   r=root;
  split=0;
  while(r->flag=='n') {  /* find the leaf */
     right=rightmost(r,obj.rect.ub[dim-1]);
     r=r->NE[right].p;
   }
  right=rightmost(r,obj.rect.ub[dim-1]);       /* find the position */

  r1=r;  right1=right;
  rightentry(r,right,&r1,&right1);
  if(obj.rect.ub[dim-1]<r1->TE[right1].g)
    dupulst(r,right,obj,&split,1);  /* duplicate the u-list divided by the upper bound */
  if(right>r->load-1)  {       /* pos is on the right of splitted leaf */
     right=right-r->load;
     r=r->pb.tl.right;
   }
  addobj(r,right,obj,&split);       /* add obj into every u-list */
  if(split==0)
   return(root);
  else              /* at least 1 split */
   return(rebuild(root));
 }


/*===== dupulst duplicates a u-list */

void dupulst(node, pos, obj, split, upper)
 page *node;      /* the node containing the roghtmost entry of the u-list */
 int pos, *split, upper;         /* pos: the rightmost object in the u-list,
```

```
                                        upper=1: dealing with the upper bound */
object obj;

{
 int cutpoint, high, left, len, g, len1, ld;
 page *node1, *node2;

 if(upper==1)
  cutpoint=obj.rect.ub[dim-1];
 else
  cutpoint=obj.rect.lb[dim-1];
 node1=node;
 g=node->TE[pos].g;
 left=pos;
 if(upper==0 && node->TE[pos].tplid!=0)
    len=1;       /* the u-list is divided by the lower bound */
 else
    len=0;
 while(node1->TE[left].g==g && node1!=0)  {
    leftentry(node1,left,&node1,&left);
    len++;              /* the length of the u-list */
  }
 ld=node->load;
 if(ld+len<=capa) {  /* has enough room */
    rshift(node,ld-1,node,ld-1+len,node,pos+1,ld-pos-1);
                        /* movw right to make room to duplicate
                        every entry */
    if(upper==1)
      dupent(node,pos,node,pos+len,len,cutpoint,0);
     else
      dupent(node,pos,node,pos+len,len,cutpoint,obj.tplid);
    node->load=ld+len;
  }
 else  {   /* need at least 1 new node */
    len1=len;  node2=node;
    do {  /* link all new nodes in */
       node1=(page *)calloc(1,sizeof(page));
       node1->pb.tl.right=node2->pb.tl.right;
       node1->pb.tl.left=node2;
       if(node2->pb.tl.right!=0)
         node2->pb.tl.right->pb.tl.left=node1;
       node2->pb.tl.right=node1;
       node2=node1;
       node1->flag='t';
       node1->load=capa;  /* the node in the middle are set full */
       len1=len1-capa;
      }
    while(len1+ld>capa);
    if(len1+ld>(capa+1)/2) {      /* fregment > half page */
                              /* all new pages are full */
       rshift(node,ld-1,node1,capa-1,0,0,ld-pos-1);
       node->load=ld-len1;
       if(upper==1)
```

```
            dupent(node,pos,node1,capa-ld+pos,len,cutpoint,0);
          else
            dupent(node,pos,node1,capa-ld+pos,len,cutpoint,obj.tplid);
        }
       else  {          /* fregment < half page */
         high=(ld+len1+capa+1)/2;
         if(high<ld-pos && node->pb.tl.right==node1)  { /* 1 new page, pos is in node */
             rshift(node,ld-1,node1,high-1,node,pos+len+1,ld-pos-1);
             node1->load=high;
             node->load=ld+len-high;
           }
          else  {         /* > 1 new node or pos is not in node, */
                          /* nodes at middle are full */
             rshift(node,node->load-1,node1,high-1,
                          0,0,node->load-pos-1);
             node1->load=high;
             node->load=(ld+len-high)%capa;
           }
         if(ld-pos-1<=high)  /* the right end is in the rightmost new node */
           if(upper==1)
             dupent(node,pos,node1,high-ld+pos,len,cutpoint,0);
           else
             dupent(node,pos,node1,high-ld+pos,len,cutpoint,obj.tplid);
          else      /* the right endi is at the left of the rightmost new node */
           if(node->pb.tl.right!=node1)
             if(upper==1)
                dupent(node,pos,node1->pb.tl.left,
                  capa+high-ld+pos-1,len,cutpoint,0);
              else
                dupent(node,pos,node1->pb.tl.left,
                  capa+high-ld+pos-1,len,cutpoint,obj.tplid);
             else
              if(upper==1)
                dupent(node,pos,node,pos+len,len,cutpoint,0);
              else
                dupent(node,pos,node,pos+len,len,cutpoint,obj.tplid);
        }
       *split=1;
    }
  }

/*===== rshift moves the right section of a node to the right */

void rshift(snode,spos,dnode,dpos,tnode,tpos,len)
 page *snode, *dnode, *tnode;    /* source, destination and terminate nodes */
 int spos, dpos, tpos, len;   /* positions and length */

 {
  if(dnode!=tnode)  {
     while(dpos>=0 && len>0)   {
       dnode->TE[dpos--]=snode->TE[spos--];
       len--;
```

```
        }
       if(len==0)
        return;
       else
        dnode=dnode->pb.tl.left;
       dpos=tpos+len-1;
     }
   while(dpos>tpos)
     dnode->TE[dpos--]=snode->TE[spos--];
 }


/*===== dupent duplicates all entries of a u-list. When duplicate a u-list
concerned with the upper bound, tplid is given to be 0; when lower
bound, tplid will be put into the u-list */

void dupent(snode,spos,dnode,dpos,len,g,tplid)
 page *snode, *dnode;       /* source and destination nodes */
 int spos, dpos, len, g, tplid;       /* positions, length, guide post and tuple id */

 {
  page *node1;
  int pos1;

  node1=snode;  pos1=spos;
  if(snode!=dnode)  {
      while(snode!=dnode) {
        while(dpos>=0)   {
          if(len>0)  {
            if(tplid == 0)  {       /* dopy an entry */
                dnode->TE[dpos].tplid
                  =node1->TE[spos].tplid;
                leftentry(node1,spos,&node1,&spos);
               }
             else  {   /* put tplid into the u-list */
                dnode->TE[dpos].tplid=tplid;
                tplid=0;
               }
            dnode->TE[dpos--].g=g;
           }
          else  {  /* copy the entries at the left in the same node */
            dnode->TE[dpos--]=node1->TE[spos];
            leftentry(node1,spos,&node1,&spos);
           }
          len--;
          if(len==0) {
            node1=snode;  spos=pos1;
           }
         }
        dpos=capa-1;
        dnode=dnode->pb.tl.left;
       }
      dpos=pos1+len;
```

```
    }
  while(len>0) {       /* for the last node of the u-list */
     if(tplid == 0) {
        dnode->TE[dpos].tplid=node1->TE[spos].tplid;
        leftentry(node1,spos,&node1,&spos);
       }
      else {
        dnode->TE[dpos].tplid=tplid;
        tplid=0;
       }
     dnode->TE[dpos--].g=g;
     len--;
    }
 }


/*===== leftentry gives the node and the position of the entry at the
left of the given entry */

void leftentry(node, pos, lnode,lpos)
 page *node, **lnode;        /* lnode: the resulting node */
 int pos, *lpos;  /* lpos: the resulting position */

 {
  if(pos>0)  {
     *lpos=pos-1;  *lnode=node;
   }
   else   {
     if(node->flag=='i')
       *lnode=node->pb.il.left;
      else
       *lnode=node->pb.tl.left;
     if(*lnode!=0)
       *lpos=(*lnode)->load-1;
    }
 }


/*===== rightentry gives the node and the position of the entry at the
right of the given entry */

void rightentry(node, pos, rnode, rpos)
 page *node, **rnode;        /* rnode: the resulting node */
 int pos, *rpos;  /* rpos: the resulting position */

 {
  if(pos<node->load-1)  {
     *rnode=node;  *rpos=pos+1;
   }
   else   {
     if(node->flag=='i')
       *rnode=node->pb.il.right;
      else
```

```
        *rnode=node->pb.tl.right;
       if(*rnode!=0)
        *rpos=0;
    }
  }

/*===== addobj adds an object to every covered u-list */

void addobj(rnode, rpos, obj, split)
 page *rnode;
 int rpos, *split;
 object obj;

 {
  page *node1, *dnode, *newnode;
  int fin, ins, rg, rg1, firstpos, lpos, objinc,
     high, low, pos1, spos, dpos;
  struct tleaf *rlf, *dlf;

  fin=0;
  firstpos=rpos;  objinc=0;
  dnode=rnode;  dpos=rpos;
  rg=rnode->TE[rpos].g+1;

  while(fin!=1)  {
     node1=rnode; pos1=rpos;
     while(rpos>=0 && fin!=1) {       /* in a node */
        if(rnode->TE[rpos].g<obj.rect.lb[dim-1])
          fin=1;          /* out of the covered u-lists */
        while(node1->TE[pos1].g==rnode->TE[rpos].g)  {
                        /* search for the left end of a u-list */
          rpos--;
          if(rpos<0) break;
         }
        if(node1->TE[pos1].tplid!=0 && fin!=1)
          objinc++;             /* covered and non-empty u-list */
        if(rnode->pb.tl.left==0 && rpos<0
             || rnode->TE[rpos].g<obj.rect.lb[dim-1])
          fin=1;                /* out of the covered u-lists */
        pos1=rpos;
       }
     rnode=dnode;
     spos=rnode->load-1;
     if(rnode->load+objinc<=capa) {   /* has enough room */
        rnode->load=rnode->load+objinc;
        rpos=pos1+objinc;
        dnode=rnode;
       }
      else  {       /* split */
        high=(rnode->load+objinc+1)/2;
        low=rnode->load+objinc-high;
```

```
    dnode=(page *)calloc(1,sizeof(page));
    *split=1;
    dnode->pb.tl.right=rnode->pb.tl.right;
    dnode->pb.tl.left=rnode;
    if(rnode->pb.tl.right!=0)
      rnode->pb.tl.right->pb.tl.left=dnode;
    rnode->pb.tl.right=dnode;
    dnode->load=high;
     dnode->flag='t';
    dpos=high-1;
    rnode->load=low;
   }
dpos=dnode->load-1;
while(firstpos<spos) {        /* the solid part on the right */
   dnode->TE[dpos]=rnode->TE[spos--];
   leftentry(dnode,dpos,&dnode,&dpos);
  }

if(rnode->TE[spos].g!=rg)  /* need a new entry */
 ins=1;
while(spos>=0 && rnode->TE[spos].g>=obj.rect.lb[dim-1])  {
   while(ins==1) {      /* put obj in */
      ins=0;
      if(rnode->TE[spos].g>=obj.rect.lb[dim-1])
        dnode->TE[dpos].g=rnode->TE[spos].g;
       else          /* for the leftmost covered u-list */
        dnode->TE[dpos].g=obj.rect.lb[dim-1];
      dnode->TE[dpos].tplid=obj.tplid;
      leftentry(dnode,dpos,&dnode,&dpos);
      if(spos>=0 && rnode->TE[spos].tplid==0
         && rnode->TE[spos].g>=obj.rect.lb[dim-1])  {
         spos--;  ins=1;
        }
      node1=rnode;  pos1=spos;
    }
   if(rnode->pb.tl.left==dnode)   /* out of the covered u-lists */
     break;
   rg=rnode->TE[spos].g;
   if(rnode->TE[spos].tplid != 0)
      dnode->TE[dpos]=rnode->TE[spos];
    else  {
        dnode->TE[dpos].g=rnode->TE[spos].g;
        dnode->TE[dpos].tplid=obj.tplid;
      }                       /* move an entry */
   leftentry(dnode,dpos,&dnode,&dpos);
   spos--;
   if(spos>=0 && rnode->TE[spos].g!=rg)
     ins=1;         /* need a new entry */
 }            /* end of a node */
rg=rnode->TE[0].g;
rnode=rnode->pb.tl.left;
if(rnode!=0)  {   /* the next node at the left */
   firstpos=rnode->load-1;
```

```
        if(rnode->TE[firstpos].g<obj.rect.lb[dim-1])
          fin=1;
        else  {
            while(rnode->TE[firstpos].g==rg)
              firstpos--;              /* the right solid part */
            objinc=0;
            rpos=firstpos;
          }
      }
    }              /* fin !=1 */
  rightentry(dnode,dpos,&rnode,&rpos);
  if(dpos>=0 && rnode->TE[rpos].g>obj.rect.lb[dim-1])
      dupulst(dnode,dpos,obj,split,0); /* the lower bound
                  seperates a u-list, duplicate the u-list */
}


/*===== duplicate duplicates the substructure */

page *duplicate(stree,prev)
 page * stree, **prev;        /* root of the source structure, current position of linked
                                  sequential list */

 {
  page *cptree, *head;
  int i;

  if(stree==0)  return(0);

  cptree=(page *)calloc(1,sizeof(page));
  cptree->load=stree->load;
  cptree->flag=stree->flag;

  switch (stree->flag)  {
    case 't':
      for(i=0;i<stree->load;i++)
        cptree->TE[i]=stree->TE[i];
      cptree->pb.tl.left=(*prev); /* link the leaf in the seq-list */
      cptree->pb.tl.right=0;
      if(*prev) (*prev)->pb.tl.right=cptree;
      (*prev)=cptree;
      return(cptree);
    case 'n':
      for(i=0;i<stree->load;i++)  {
        cptree->NE[i].g=stree->NE[i].g;
        cptree->NE[i].p=duplicate(stree->NE[i].p, prev);
       }
      return(cptree);
    case 'i':
      for(i=0;i<stree->load;i++)  {
        cptree->IE[i].g=stree->IE[i].g;
        head=0;
        cptree->IE[i].p=duplicate(stree->IE[i].p, &head);
```

```
        }
        cptree->pb.il.left=(*prev); /* link the leaf into the seq-list */
        cptree->pb.il.right=0;
        if(*prev) (*prev)->pb.il.right=cptree;
        (*prev)=cptree;
        return(cptree);
    }
  }

/*===== rebuild rebuilds a TP-tree in which split has happened */

page *rebuild(root)
  page *root;       /* the root of an TP-tree */

  {
    typedef struct lvlchn  {
        page *node;
        struct lvlchn *next;
      } lvlchn;
    int i, count;
    page *head, *head1;
    lvlchn *chain, *chp, *chainhy, *chphy;

    head=head1=root;
    while(head->flag!='t') {  /* find the leftmost leaf */
        head1=head->NE[0].p;
        if(head1->flag!='t')
          for(i=1;i<head->load;i++)
            free(head->NE[i].p);
        free(head);
        head=head1;
      }

    count=1;
    chain=(lvlchn *)calloc(1,sizeof(lvlchn));
    chain->node=head; chain->next=0;
    chp=chain;              /* a chain covers a level of nodes */
    while(head1->pb.tl.right!=0)  {       /* count the node on the level */
        head1=head1->pb.tl.right;
        count++;
        chp->next=(lvlchn *)calloc(1,sizeof(lvlchn));
        chp=chp->next; chp->node=head1;  chp->next=0;
      }

    while(count>1)  {
        count=1;
        head=head1=(page*)calloc(1,sizeof(page));
        head->flag='n'; head->load=0;
        chainhy=(lvlchn*)calloc(1,sizeof(lvlchn));
        chainhy->node=head;  chainhy->next=0;  chphy=chainhy;
        while(1) {
          head1->NE[head1->load].p=chain->node;
```

```
          if(chain->node->flag=='n')
            head1->NE[head1->load].g
                =chain->node->NE[0].g;
          else
            head1->NE[head1->load].g
                =chain->node->TE[0].g;
          head1->load++;
          chp=chain;  chain=chain->next; free(chp);
          if(chain==0)   break;
          if(head1->load==fanout)    {
              chphy->next=(lvlchn*)calloc(1,sizeof(lvlchn));
              count++;
              chphy=chphy->next,  chphy->next=0;
              chphy->node=head1=(page*)calloc(1,sizeof(page));
              head1->flag='n';  head1->load=0;
            }
         }
       chain=chp=chainhy;
     }
   free(chain);
   return(head);
  }

/*===== seqlst lists a sequential list of a TP-tree */

void  seqlst(head)
  page *head;      /* the head of the list */

  {
   int c, i;

   c=0;
   ofile=fopen("seq","a");
   while(head!=0)  {
       if(c>55)  { fprintf(ofile,"\n"); c=0; }
       fprintf(ofile," $");
       for(i=0;i<head->load;i++)  {
         fprintf(ofile,"%d %d/",
                head->TE[i].g,head->TE[i].tplid);
         c=c+10;
         }
       head=head->pb.tl.right;
     }
   fprintf(ofile,"\n\n");
   fclose(ofile);
  }

/*===== tplst lists the sequential lists in a tree */

void tplst(root)
  page *root;

  {
```

```
  int pos=0;

  if(root==0)  {
    ofile=fopen("seq","a");
    fprintf(ofile," $.. ");
    fclose(ofile);
   }
   while(root->flag=='n')
    root=root->NE[0].p;
   seqlst(root);
   if(root->flag=='i')
    while(root)  {
      tplst(root->IE[pos].p);
      rightentry(root,pos,&root,&pos);
     }
 }


/*===== search searches the area of "rect" in the tree rooted at "root" on
the stage numbered with "stage" */

objlist *search(root, rect, stage)
 page *root;
 rectangle rect;
 int stage;
{ int rtmost, leftend, g1, rtpos;
 page * rtroot;
 objlist *coll, *obj;

 coll=(objlist*)calloc(1,sizeof(objlist));
 coll->tplid=0;  coll->next=0;
 if(root==0) return(coll);
 if(root->flag!='n')
  rtmost=rightmost(root, rect.ub[stage-1]);
 else
   while(root->flag=='n')  {
     rtmost=rightmost(root, rect.ub[stage-1]);
     root=root->NE[rtmost].p;
    }
 rtmost=rightmost(root,rect.ub[stage-1]);
 if(root->flag=='t')  {
     leftend=0;
     do {
       if((g1=root->TE[rtmost].g)<=rect.lb[stage-1])
         leftend=1;
       if(root->TE[rtmost].tplid != 0) {
          obj=(objlist*)calloc(1,sizeof(objlist));
          obj->next=0;
          obj->tplid=root->TE[rtmost].tplid;
          accobject(coll,obj);        /* not a dummy */
         }
       leftentry(root,rtmost,&root,&rtmost);
       }
```

```
      while(leftend==0||g1==root->TE[rtmost].g);
    }
  else /* 'i' */     {
      accobject(coll, search(root->IE[rtmost].p,rect, stage+1));
      while(root->IE[rtmost].g>rect.lb[stage-1]) {
        leftentry(root, rtmost, &root, &rtmost);
        accobject(coll, search(root->IE[rtmost].p, rect, stage+1));
       }
     }
  return(coll);
}


/*===== prtolst prints the list of objects */

void prtolst(head)
 objlist *head;
{
 objlist *p;

 ofile=fopen("seq","a");
 p=head->next;
 fprintf(ofile,"\nThe intersecting objects:");
 while(p) {
   fprintf(ofile,"  %d",p->tplid);
   p=p->next;
  }
 fprintf(ofile,"\n");
 fclose(ofile);
}


/*===== accobject accumulates the acsendently ordered object list
'cl1' by the objects in the object list 'cl2' */

void accobject(col1, col2)
  objlist *col1, *col2;
{
 objlist *obj, *pos1, *pos2, *obj1;
 int found;

 obj=col2;
 while(obj) {       /* for every object in col2 */
   pos1=col1; found=0; obj1=obj->next;
   while(!found)  {
     pos2=pos1; pos1=pos1->next;
     if(pos1==0 || pos1->tplid>obj->tplid)
       found=1;
     }
   if(pos2->tplid!=obj->tplid)        /* insert */  {
     obj->next=pos1;
     pos2->next=obj;
     }
```

```
  obj=obj1;
  }
}

/*===== del deletes an object from the tree on the 'stage' and from
its substructures, return the new root of the tree */

page *del(root, obj, stage)
 page *root;
 object obj;
 int stage;
{
 if(stage<dim)
  if(root->flag=='n')   /* non-leaf of an IP-tree */
   deln(root, obj, stage);
 else                   /* leaf of an IP-tree */
   deli(root, obj, stage);
 else                   /* TP-tree */
  root=delt(root, obj);
 return(root);
}


/*===== delete deletes an object from the IDP rooted at 'root' */

page *delete(root, obj, stage)
 page *root;
 object obj;
 int stage;
{
 page *r;

 r=root=del(root, obj, stage);
 if(root->load==1 && root->flag=='n')  {
  r=root->NE[0].p;
  free(root);
  }
 return(r);
}


/*===== deln deletes from a structure rooted at a non-leaf of an IP-tree */

void deln(root, obj, stage)
 page *root;
 object obj;
 int stage;
{
 int i, rtmost;
 page *p1;

 rtmost=rightmost(root, obj.rect.ub[stage-1]);
 while(rtmost >= 0 && root->NE[rtmost].g >= obj.rect.lb[stage-1])  {
```

```
  root->NE[rtmost].p=p1=del(root->NE[rtmost].p, obj, stage);
  if(p1->load<(capa+1)/2)
   rearrange(root, &rtmost);
  rtmost-=1;
 }
 for(i=0; i<root->load; i++) {
  p1=root->NE[i].p;
  root->NE[i].g= (p1->flag=='n')?
   p1->NE[0].g:
   p1->IE[0].g;
 }
}


/*===== deli deletes an object from the trees pointed to by entries of
a leaf node of an IP-tree on the 'stage' */

void deli(node, obj, stage)
 page *node;
 object obj;
 int stage;
{
 int rtmost, i, lrtmt=0, rrtmt=0;
 page *lrt=0, *rrt=0, *p1;

 rtmost=rightmost(node, obj.rect.ub[stage-1]);
 while(rtmost>=0 && node->IE[rtmost].g >= obj.rect.lb[stage-1]) {
  node->IE[rtmost].p =p1 =del(node->IE[rtmost].p, obj, stage+1);
  if(p1->load==1 && p1->flag=='n') {       /* an unneccesary node */
   node->IE[rtmost].p=p1->NE[0].p;
   free(p1);
   }
  rightentry(node, rtmost, &rrt, &rrtmt);
  if(rrt!=0 && rrt->IE[rrtmt].g==obj.rect.ub[stage-1])   /* right boundary */
   if(sameobj(node, rtmost, rrt, rrtmt, stage))  {
    freesub(node->IE[rtmost].p, stage);
    if(node==rrt) {   /* the edge is withih the 'node' */
      node->IE[rtmost].p=node->IE[rrtmt].p;
      rmventry(node,rrtmt);
     }
    else {        /* the edge is between 'node' and 'rrt' */
      rrt->IE[0].g=node->IE[rtmost].g;
      node->load-=1;
     }
    }
   if(node->IE[rtmost].g==obj.rect.lb[stage-1]){  /* the left boundary */
    leftentry(node, rtmost, &lrt, &lrtmt);
    if(lrt!=0)
      if(sameobj(node, rtmost, lrt, lrtmt, stage)) {
       freesub(node->IE[rtmost].p, stage);
       rmventry(node,0);
       }
    }
```

```
    rtmost--;
   }
}

/*===== delt deletes an object from a TP-tree rooted at 'root' */

page *delt(root, obj)
 page *root;
 object obj;
{
 int rtmost, rrtmt=0, cutpoint, shtg;
 page *rt, *rrt=0;
 objlist *ol1, *ol2;

 rt=root;
 while(rt->flag!='t')  {
  rtmost=rightmost(rt,obj.rect.ub[dim-1]);
  rt=rt->NE[rtmost].p;
  }
 rtmost=rightmost(rt,obj.rect.ub[dim-1]);
 rightentry(rt, rtmost, &rrt, &rrtmt);
 if(rrt!=0) {        /* the right boundary is within the space */
  ol1=collulst(rt, rtmost, obj.tplid, -1);
  ol2=collulst(rrt, rrtmt, obj.tplid, 1);
  if(samelst(ol1,ol2))          /* remove the redundent uniform list */
    rmvulst(rrt,rrtmt,rrt->TE[rrtmt].g);
  }
 while(rt!=0 && rt->TE[rtmost].g>=obj.rect.lb[dim-1])  {
              /* scan and remove all entries of the obj, leftward */
  if(rt->TE[rtmost].tplid==obj.tplid)
    rmventry(rt, rtmost);
  if(rtmost==0)
    if(rt->load<(fanout+1)/2 && (rrt=rt->pb.tl.right)!=0)
      if(rt->load+rrt->load <= fanout) {
          mvleft(rrt,0,rt,rt->load,rrt->load);
          rt->pb.tl.right=rrt->pb.tl.right;
          if(rrt->pb.tl.right)
            rrt->pb.tl.right->pb.tl.left=rt;
          free(rrt);
         }
      else {
         shtg=(fanout+1)/2-rt->load;
         mvleft(rrt,0,rt,rt->load,shtg);
         mvleft(rrt,shtg,rrt,0,rrt->load);
        }
  leftentry(rt, rtmost, &rt, &rtmost);
  }
 if(rt!=0) { /* the left boundary of the object */
  rightentry(rt, rtmost, &rrt, &rrtmt);
  ol1=collulst(rt, rtmost, obj.tplid, -1);
  ol2=collulst(rrt, rrtmt, obj.tplid, 1);
  if(samelst(ol1, ol2))
    rmvulst(rrt,rrtmt,rrt->TE[rrtmt].g);
```

```
  }
 return(rebuild(root));
}


/*===== rmvulst removes a uniform list, the leftmost entry of which is
at 'pos' in 'node' and with the guidepost 'egde' */

void rmvulst(node, pos, edge)
 page *node;
 int pos, edge;
{
 page *node1;
 int pos1, shtg, i;

    while(node!=0 && node->TE[pos].g==edge)  {
     rmventry(node, pos);
     if(pos == node->load || pos<node->load && node->TE[pos].g!=edge) {
                    /* no more entries in the node should be removed */
         node1=node;  pos1=pos-1;  pos=0;
         node=node->pb.tl.right;
       }
     if((node->TE[pos].g!=edge || pos1==node1->load-1) && node!=0) {
                    /* check the load to shift of merge */
         if(node1->load<(fanout+1)/2)   {
          if(node1->load+node->load<=fanout) {
              mvleft(node,0,node1,node1->load,node->load);
              node1->pb.tl.right=node->pb.tl.right;
              if(node1->pb.tl.right)
                node1->pb.tl.right->pb.tl.left=node1;
              free(node);
            }
          else  {
              shtg=(fanout+1)/2-node1->load;
              mvleft(node,0,node1,node1->load,shtg);
              mvleft(node,shtg,node,0,node->load-shtg);
            }
          node=node1;  pos=pos1+1;
         }
       }
    }
}


/*===== sameobj compares two sets of objects in the structures pointed to
by the 'entry1' in 'rt1' and 'entry2' in 'rt2'. both 'rt1' and 'rt2'
are on the stage numbered by 'stage' */

int sameobj(rt1, entry1, rt2, entry2, stage)
 page *rt1, *rt2;
 int entry1, entry2, stage;
{
 int i;
```

```
 objlist *ol1, *ol2;
 rectangle space;

for(i=0; i<dim; i++)
 { space.lb[i]=0; space.ub[i]=bound; }
if(rt1->flag=='n') {
  ol1=search(rt1->NE[entry1].p, space, stage);
  ol2=search(rt2->NE[entry2].p, space, stage);
 }
else {
  ol1=search(rt1->IE[entry1].p, space, stage);
  ol2=search(rt2->IE[entry2].p, space, stage);
 }
return( samelst(ol1,ol2));
}


/*===== samelst determines whehter two assendent linked list of objects
contain the same set of objects */

int samelst(ol1,ol2)
 objlist *ol1, *ol2;
{
 int same=1;

 while(ol1!=0 && ol2!=0 && same) {
  if(ol1->tplid!= ol2->tplid)
    same=0;
  ol1=ol1->next;
  ol2=ol2->next;
 }
 if(ol1==0 && ol2==0 && same)
  return(1);
 else
  return(0);
}


/*===== rmvenrtry removes the 'entry' from the 'node' */

void rmventry(node, entry)
 page *node;
 int entry;
{
 int i;

 node->load--;
 switch(node->flag) {
   case 't':
     for(i=entry; i<node->load; i++)
       node->TE[i]=node->TE[i+1];
     break;
```

```
   case 'i':
     for(i=entry; i<node->load; i++)
       node->IE[i]=node->IE[i+1];
     break;
   case 'n':
     for(i=entry; i<node->load; i++)
       node->NE[i]=node->NE[i+1];
     break;
  }
}


/* move 'len' entries from the position of 'spos' in the node of
'snode' to the position of 'dpos' in the node of 'dnode'. 'dpos'
is on the left of 'spos' */
void mvleft(snode, spos, dnode, dpos, len)
 page *snode, *dnode;
 int spos, dpos, len;
{
 int i;

 switch (snode->flag)  {
    case 'i':
     for(i=0; i<len; i++)
       dnode->IE[dpos+i]=snode->IE[spos+i];
     break;
    case 'n':
     for(i=0; i<len; i++)
       dnode->NE[dpos+i]=snode->NE[spos+i];
     break;
    case 't':
     for(i=0; i<len; i++)
       dnode->TE[dpos+i]=snode->TE[spos+i];
     break;
  }
 if(snode==dnode)
   dnode->load=dpos+len;
 else  {
   dnode->load+=len;
   snode->load-=len;
   }
}


/*===== freesub frees all storage of the subtree rooted at  'node' */

void freesub(node)
 page *node;
{
 int i;

 switch(node->flag)  {
```

```
  case 't':
   free(node);
   break;
  case 'i':
   for(i=0;i<node->load; i++)
    freesub(node->IE[i].p);
   free(node);
   break;
  case 'n':
   for(i=0;i<node->load; i++)
    freesub(node->NE[i].p);
   free(node);
 }
return;
}


/*===== collulst collects objects in a uniform list, leftward if dir<0
rightward otherwise, from the 'entry in the 'node'. The object
with tplid equal to 'id' is excluded from the result. */

objlist *collulst(node, entry, id, dir)
 page *node;
 int entry, id;
 int dir;

{
 int g;
 objlist *ol1, *ol2;

 g=node->TE[entry].g;
 ol1=(objlist*)calloc(1,sizeof(objlist));
 ol1->tplid=0;  ol1->next=0;
 while(node!=0&&&node->TE[entry].g==g)  {
  if(node->TE[entry].tplid!=id)   {
   ol2=(objlist*)calloc(1,sizeof(objlist));
   ol2->tplid=node->TE[entry].tplid;
   ol2->next=0;
   accobject(ol1,ol2);
   }
  if(dir<0)
   leftentry(node, entry, &node, &entry);
  else
   rightentry(node, entry, &node, &entry);
 }
 return(ol1);
}


/*===== rearrange merges the node pointed to by 'entry[subtree]' in 'node'
with a neighbor node or move some entries from a neoghbor into the node */

void rearrange(root, subtree)
```

```
page *root;
int *subtree;
{
int rload, lload, load, i, shtg;
page *lnbr, *rnbr, *chd;

if(root->NE[*subtree].p->load>=(capa+1)/2)
  return;
lnbr=rnbr=0;
if(root->load>*subtree+1) {       /* has a right neighbor */
  rnbr=root->NE[*subtree+1].p;
  rload=rnbr->load;
  }
if(*subtree>0) { /* has a left neighbor */
  lnbr=root->NE[*subtree-1].p;
  lload=lnbr->load;
  }
chd=root->NE[*subtree-1].p;
load=chd->load;

if(rnbr!=0&&load+rload<=capa) { /* merge with the right neighbor */
  mvleft(rnbr,0,chd,chd->load,rnbr->load);
  if(chd->flag=='i') {
    chd->pb.il.right=rnbr->pb.il.right;
    if(rnbr->pb.il.right) rnbr->pb.il.right->pb.il.left=chd;
    }
  free(rnbr);
  return;
  }
if(lnbr!=0 && load+lload<=capa) { /*merge with the left neighbor */
  mvleft(chd,0,lnbr,lnbr->load,chd->load);
  if(chd->flag=='i') {
    lnbr->pb.il.right=chd->pb.il.right;
    if(chd->pb.il.right) chd->pb.il.right->pb.il.left=lnbr;
    }
  rmventry(root,*subtree);
  (*subtree)++;
  free(chd);
  return;
  }

shtg=(capa+1)/2-load;             /* shortage of entries */
if(rnbr) {        /* move some entries from the right neoghbor */
  mvleft(rnbr,0,chd,chd->load,shtg);
  mvleft(rnbr,shtg,rnbr,0,rnbr->load);
  return;
  }

if(lnbr) { /* move some entries from the left neighbor */
  if(chd->flag=='n') {
    for(i=load-1; i>=0; i--)
      chd->NE[i+shtg]=chd->NE[i];
    for(i=0; i<shtg; i++)
```

```
      chd->NE[i]=lnbr->NE[lload-shtg-1+i];
     }
    else   {
     for(i=load-1; i>=0; i--)
      chd->IE[i+shtg]=chd->IE[i];
     for(i=0; i<shtg; i++)
      chd->IE[i]=lnbr->IE[lload-shtg-1+i];
     }
    lnbr->load-=shtg;  chd->load+=shtg;
    *subtree+=1;
    return;
    }
}
```

VITA

Xiaoming Cheng

Candidate of the Degree of

Master of Science

Thesis: THE INDEX BY DIMENSIONAL PROJECTION – AN INDEX SUPPORTING SEARCH FOR SPATIAL OBJECTS BY REGION

Major Field: Computer Science

Biographical:

Personal Data: Born in Shanghai, China, January 4, 1950, the son of Wang Cheng and Junsu Qian.

Education: Received Bachelor of Science Degree in Computer Science from Shanghai Jiao Tong University at Shanghai, China in February, 1982; completed requirements for the Master of Science degree at Oklahoma State University in May, 1991.

Professional Experience: Lecturer, Engineering College, Shanghai University, February, 1982 to March, 1989.