# $G^+$-TREE: A SPATIAL INDEX STRUCTURE

By

HUNG-CHI SU

Bachelor of Science

Chen-Kung University

Tainan, Taiwan

1987

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the degree of
MASTER OF SCIENCE
July, 1999

G$^+$-TREE: A SPATIAL INDEX STRUCTURE

Thesis Approved:

H. Lu
Thesis Adviser

J. E. Hed

Wayne B. Powell
Dean of the Graduate College

ACKNOWLEDGEMENTS

I would like to express my appreciation to my thesis advisor, Dr. Huizhu Lu to thank her guidance and support over the past few years. I would also like to give thanks to my committee members, Dr. George E. Hedrick and Dr. K. M. George, for their advice and willing to serve on my graduate committee.

I would also like to give my special thank my family. Without their understanding and support, it is impossible for me to work on the degree. Moreover, I wish to express my sincere gratitude to my friends who gave helps and provided suggestions for this thesis and stood by me during the past years.

Finally, I would also like to thank the faculty and staff of the Computer Science Department who have helped me so much during these last few years of study.

TABLE OF CONTENTS

LIST OF FIGURES

# CHAPTER I

## INTRODUCTION AND MOTIVATION

Traditionally, the data structures for indexing a data file are for one-dimensional space only. This one-dimensional space is either a single attribute or composite attributes with specific order. As prominent as B-tree family (B-tree, $B^+$-tree and B*-tree), this type of data structures is not only efficient but also popular for query in one-dimensional data space. For instance, we want to search the names of all students whose I.D. numbers are between 123456789 and 213456789. To solve this one-dimensional problem, a B*-tree data is adopted by indexing I.D. numbers. Then, the result is obtainable with a few disk accesses.

However, a multi-dimensional query is more complicated. For example, finding the students whose I.D. numbers are between 123456789 and 213456789 as well as birthdays are between 05/01/70 and 05/01/71 is a two-dimensional range query. To solve this problem, we need two individual keys indexed in a traditional method. In other words, two indexes of I.D. numbers and birth dates have to be set, respectively. Then, use one index to narrow down the size of the student set. After narrowing the size, we can filter out those have met the second searching criteria. The other way is to maintain a composite index on I.D. and birth date; however, neither of the above methods is favorable enough. In addition, we may need to maintain several key indexes for a data set. It takes a great deal of space and time to accomplish.

Since the traditional index data structures are not effective enough for indexing multi-dimensional data, some methods were proposed to improve the index technique. One method used to index multi-dimensional data is named multi-attribute hashing [ROT74]. It employs a composed attribute value for the hash key and hash table to store the point data. But, this method is not useful for range queries even though it is good for exact match queries.

Two other disk-based data structures, the KDB tree (K dimensional B tree) [ROB81] and the Grid file [NIE84], were proposed to index large multi-dimensional data for data query, including range queries. The former, KDB tree, divides the entire k-dimensional data space into small k-dimensional regions and hyper-rectangles. Then, the region hyper-rectangles are assembled in the shape of a tree which structure is similar to a B-tree. Each entry of a leaf node points to a corresponding data tuple, while each entry of a non-leaf node points to the lower level nodes that are subregions within the entry region and are disjoint. The latter, Grid file, is a data structure partitioning the data space into grid structures. Each dimension of the data space is split into non-uniform space intervals. All of the physical entries of grids are maintained in a directory. The Grid file guarantees that at most two disk accesses are needed for retrieving a record, one access for directory and the other for the record. On the other hand, the disadvantage of Grid file is that the directory may be very large and the expense of directory maintenance may be very high.

To achieve the goal of high performance and low maintenance cost on indexing multi-dimensional data, the G-tree (Grid Tree)[KUM94] was thus proposed. It combines the features of both grid file and B-tree. The foundation of the G-tree is to split the overflowed partitions into two equal sized subpartitions along one coordinate. If one

subpartition still overflows (i.e., the other is empty), the overflowed subpartition has to be split again along the consequent coordinate. This procedure continues splitting subpartitions till the point data are distributed into two subpartitions. (Distribution does not matter as long as the subpartitions are not overflowed.) A unique number assigned to each subpartition is defined as a total order number sequence. Hence, all unique numbers (partition numbers) build a B-tree together. Since the G-tree manages the data in terms of partition, it can retain the property of spatial locality for some data. Therefore, this data structure, which maps the partitions into a total order sequence and organizes the numbers to be a B-tree, results the G-tree in good performance of data retrieval. Nevertheless, the possibility of poor space utilization of leaf nodes (only one entry per leaf node) entails the G-tree unsuitable for very large non-uniform databases, especially, for the range query. The other drawback in this data structure lies in the algorithm of range query. It is unrealistic that the data structure can keep the spatial locality of all data, as the locality will not consist of those data close to the spatial location but far in the grid number sequence after hundreds of splitting. It is, therefore, reasonable to perform a range query simply by sequentially searching neighbors of the leaf nodes after the first grid is found because there could be a great deal nodes in the neighborhood that are out of the query range.

Moreover, two other data structures have been proposed for better space utilization to eliminate the costly maintenance of the data indexes and to improve the querying performance. They are GBD-tree and BANG file. Both are developed for balancing the distribution of point data between the subpartitions when overflow occurs. These schemes have obviously improved the space utilization. (Note that GBD-tree guarantees balance

between split leaf nodes rather than non-leaf nodes.) In the worst case, both of GBD-tree and BANG-file have nearly one third of the capacity of the leaf nodes in respect of space utilization. The drawback in these two data structures is the poor performance on range query if data can be dynamically added to or deleted from the databases. That is, the spatial locality for the data is lost.

To improve the performance and to reduce the maintenance expense on query in the multi-dimensional database is the core consideration of this thesis. Among those data structures mentioned above, G-tree is the simplest and has good performance on indexing the multi-dimensional data. However, the worst case of storage utilization makes G-tree really unsatisfactory in regard to query performance and maintenance expense in the extreme non-uniformly data distribution situation. On the other hand, the optimistic algorithm of range query is also a significant drawback. Thus, we would like to eliminate the poor performance and speed up the algorithm of range query by improving the storage utilization of the G-tree. Therefore, the author introduces a new data structure, the $G^+$-tree, to increase the space utilization and to expedite the range query. Unlike G-tree, the $G^+$-tree distributes the point data evenly to two sub-partitions when decomposing a partition. Therefore, $G^+$-tree guarantees that the space utilization reaches 50% or higher even though under the circumstance of extreme non-uniformly data distribution.

$G^+$-tree is given the name because of being derived from G-tree with balanced distribution of point data between partitions. The $G^+$-tree employs a special and simple method for decomposing partition:

1. Decompose the whole partition into several grids till point data can be evenly distributed between two groups.

2. Group the lower grids with continuous grid numbers and upper ones into two partitions with evenly point data distribution.

Hence, each partition is assigned with a number same as the smallest constituent grid number and then, is inserted into the tree. The partitions with the assigned numbers have the property of total order as no any grid can be placed inside of two different partitions. The partitions also keep the property of spatial locality within themselves. Thus, we can eliminate the G-tree's problem of the low space utilization without degrading much the query performance.

This thesis is organized as follows. Section 1 begins with the introduction. Section 2 reviews the data structures proposed to index the multi-dimensional data. Section 3 introduces the $G^+$-tree and its algorithms. Section 4 is for performance analyses between G-tree and $G^+$-tree. Section 5 is the conclusion.

# CHAPTER II

## LITERATURE REVIEW

In this section, the literature on or related to multi-dimensional indexing method will be discussed briefly. Since this thesis is trying to construct a data structure to eliminate the drawback of G-tree in the worst case, we will discuss more on the G-tree data structure.

To efficiently manage multi-dimension objects, a database system requires an effective mechanism to index the spatial objects in order to get a high performance on accessing the data. However, the traditional one-dimensional indexing method is not suitable for the multi-dimensional data; therefore, a new data structure is needed.

Many data structures are proposed to manage the multi-dimensional point data; and all of them allocate in the secondary storage in consideration of the huge data needed to be stored. The access to the secondary memory, usually a disk, is an important factor to the performance of the system. Hence, the index nodes or bucket nodes are usually designed to be the size of disk pages. Furthermore, the fan-out of a node (the number of pointers to the subnodes) should not be too small for tree-structured system, otherwise, the tree becoming much deeper means that much more disk I/Os for data access and more time for the operations are resulted. As the time of disk I/O dominates that of other factors, the time of data retrieval proportions to the number of disk I/O for accessing the data.

The G$^+$-tree is a data structure with grids; therefore, in this section, grid-related data structures are discussed in more detail. The size of a bucket node is assumed to be same size for the grid-related data structures.

## Grid File

Grid file [NIE84] is one of the data structures that partition the data space into disjoint grids by splitting each dimension into several space intervals. (The space intervals constitute hyper-rectangles that are referred to be grids in this thesis.) There exists a directory maintaining grid entries. Each grid is designed to contain less data than a disk page does. The later (disk page) contains the data enclosed with the corresponding grid. In other words, every grid has a maximum number of stored data; however, different elements may point to the same disk page if the total number of the data is less than the capacity of a disk page.

The grid file structure is proposed for data retrieval by at most two disk accesses and for efficient range query execution. To achieve these two purposes, it maintains a dynamic grid directory of the grid pages for access. The directory consists of two parts. One is a k-dimensional array which contains the entries for all the grids; the other is a set of k one-dimensional arrays called linear scales which define the domain of each attribute in order to indicate the grid directory elements that overlap the query range.

The first purpose of the grid file (i.e., to retrieving data by at most two disk access) can be achieved by performing one disk access for the grid entry in the directory and the other disk access via the grid entry for the grid page which keeps the data. The second purpose (i.e., processing range query efficiently) can be met in terms of the linear scales (second part of the grid directory) to indicate the grid directory elements that overlap the query.

However, the maintenance of the grid directory can be very expensive and the size of the directory will be very large if the distribution of the data is non-uniform. Grid file,

therefore, may not be an effective method for a very large database, especially, is not for that with non-uniformly distributed data.

## G-Tree

The G-tree [KUM94] is one of the data structures for multi-dimensional data access. It divides the data space cyclically along each dimension into disjoint partitions. The number of point data in a partition is between one and the capacity of a bucket node (the capacity of a bucket node can be assumed as *max_num*). A unique number is assigned to each subpartition which results from the original (parent) partition. Each unique number is derived from the parent partition number concatenated with either '0' if the sub-partition is the lower part along the splitting dimension or '1' if it is the upper part on the splitting dimension. This partitioning scheme is shown as Fig. 1. The partition numbers are maintained in a B-tree. Because of taking advantage of B-tree's structure, the G-tree performs well on data retrieval.

The features of the G-tree are: (quoted from [KUM94])

- The data space is divided into non-overlapping partitions of rectangles (hyper-rectangle) in various size.
- Each partition is assigned with a unique partition number.
- Partition numbers are defined in a total order then stored in a B-tree.
- Empty partitions are not stored in the tree to save storage space.

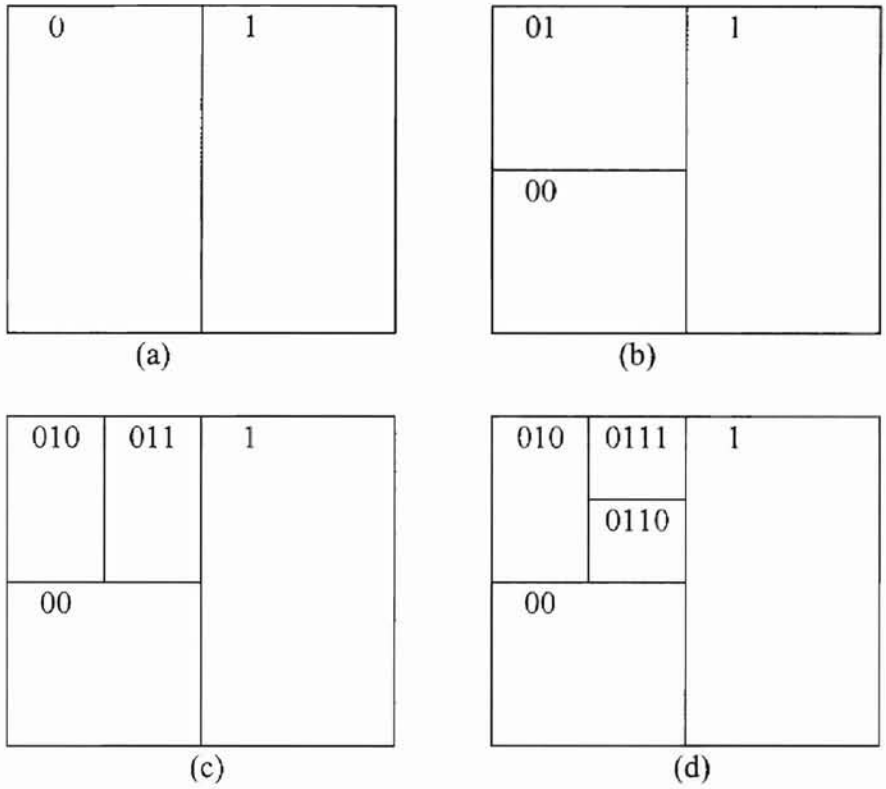


Fig. 1 Partitioning scheme of G-tree

(a) First, partition along the coordinate 1. Lower part (left in this Fig.) is assigned with '0' and higher part (right) with '1'.

(b) Partition along the coordinate 2. Lower part is padded with a '0' and higher part with a '1'.

(c) Partition along the coordinate 1 again.

(d) Partition along the coordinate 2 again.

Storage utilization is the major problem of G-trees. The G-tree will elicit very dissatisfying storage utilization for large non-uniformly distributed data since they might allocate as few as only one point data in a partition. For example, the G-tree splitting algorithm splits an overflowed partition which is *max_num* + 1 into two sub-partitions that may contain *max_num* point data in one sub-partition and only one point datum in the other. After several splits like this situation, there will be many partitions that contain only one datum; meanwhile, the storage utilization of the bucket nodes will be close to 1/*max_num* (though it does not happen often). That is, if *max_num* is 25, the space utilization is nearly 4%. It is terrible for a huge database.

The disadvantage of storage utilization causes the G-tree unsuitable for managing a very large non-uniformly distributed multidimensional point data.

## GBD-Tree

The GBD-tree (General BD-tree) [OHS90] was proposed for the multi-dimensional spatial data rather than the point data. However, it can be a good algorithm for multidimensional point data as GBD-tree uses one point (the centroid) to represent the spatial object.

This type of data structure is similar to G-tree in respect of its partition number scheme. The distinction is that the partition numbers (bit strings) of the GBD are concatenated with '*' for each partition to denote the end of the bit string. This partitioning scheme is shown as Fig. 2. Note that the partition may enclose some other partitions, i.e., partitions may be nested inside the others. For example, partition 00* is nested inside partition 0*.

The most significant differences between GBD-tree and G-tree are the modes of splitting a partition and constructing an index-tree. On the one hand, the partitions of a G-tree are disjoint and accommodate the number of point data between one and *max_num* while the partitions of GBD-tree are not disjoint and accommodate a balanced number of point data. The balanced number in a GBD-tree is any number between one third and two thirds of capacity (*max_num*). The scheme of the splitting partition leads GBD-tree to a better data distribution between bucket nodes (page nodes) than G-tree does. On the other hand, the G-tree holds the partition numbers in a B-tree, but the GBD-tree adopts a hierarchical structure with ancestor-descendant-relationship. In short, the partition number of the entry in the parent nodes prefixes the partition number in the child nodes.

Although the GBD-tree can obtain better balance for the number of points in partitions, it does not guarantee balance for the non-leaf nodes. GBD-tree suffers the

similar problem to what G-tree has though, its balance can guarantee to be one third of *max_num* +1 in the page nodes in the worst case. (This may not be a good structure in the view of balanced tree.) Besides, it may not be a good structure for range query since its structure is nested.

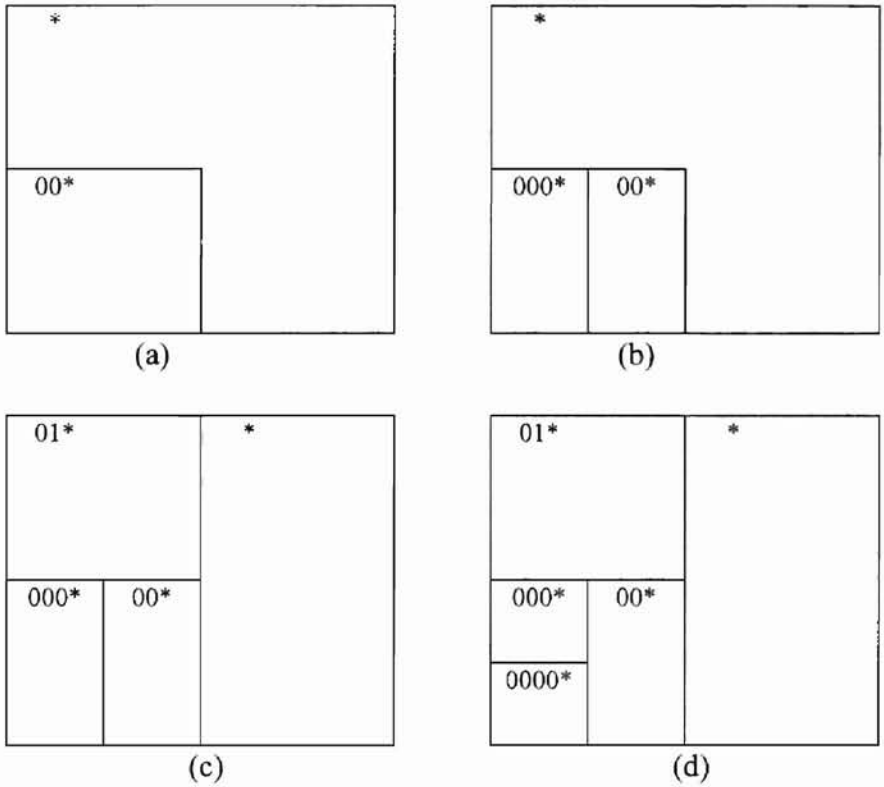


Fig. 2 Numbering scheme of GBD-tree

(a) First, partition along the coordinate 1 and coordinate 2 till data are distributed evenly. Inner (lower left) part is assigned with '00*' (because it is in the lower and the left part) and outer part, '*', remains unchanged.

(b) Partition in the '00*' region. Inner part is padded with '0' and outer part remains unchanged.

(c) Partition in '*' region. Then, the right upper grid is split from the '*' and assigned with '01*' because it is in the right and upper location.

(d) Partition in '000*' region. The inner part is lower, so '0' is padded. The outer part remains unchanged

## BANG File

The BANG (Balanced and Nested Grid) file [FRE87] is an interpolation grid file. It partitions the data space into block regions by successive binary division. The union of all the subpartitions must span the data space. Each block region is identified by (r, *l*), which is the region number. The *l* is the granularity or level number. The numbering scheme for grid regions is shown as Fig. 3.

The number of point data in a partition is restricted to be *max_num* as other tree structures do. Therefore, the partition splits when the number of point data is over *max_num*. The splitting method is trying to split into two regions with equivalently distributed number of point data by successive binary division. Like the GBD-tree, BANG file permits one partition nested inside the others. The worst case in data distribution may be one third of *max_num*. So is GBD-tree.

The regions are organized as tree structure for index. Non-leaf nodes treat all of their children in the same way as the leaf nodes do the point data. Then, the balancing algorithm can be applied to both of leaf and non-leaf nodes. In short, all the nodes in the tree structure are better balanced in space utilization. As leaf nodes are always updated upwards, BANG file has a compact and balanced structure similar to B-tree.

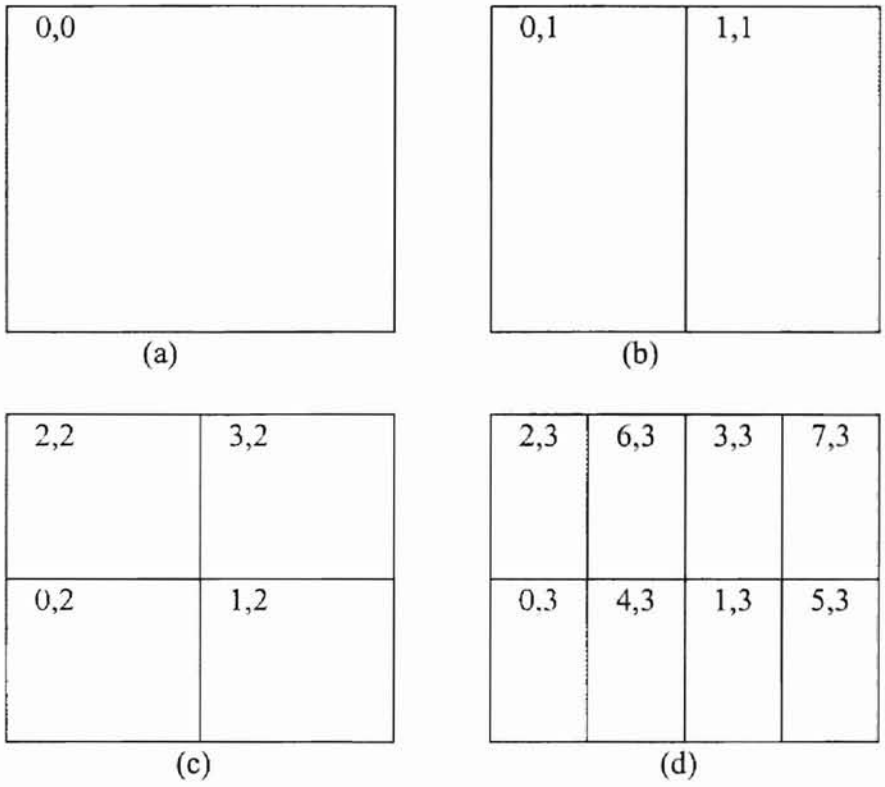


Fig. 3 Numbering scheme of grid regions for BANG file

(a) No any splitting. The region number is 0 and the level number is 0, too. That is, the $(r, l)$ is (0,0).

(b) First split is undertaken along the dimension 1. The region number is 0 for the lower part (left in this case) and 1 for upper (right) part. That is, (0, 1) and (1, 1) for each $(r, l)$.

(c) Second split is undertaken along the dimension 2. Hence, the level number is 2 and the region numbers are inherited from itself. The region number is added with $d^{l-1}$ where d is the number of dimensions. So, 0 splits to two region numbers, 0 and $0+2^{2-1}=2$.

(d) Third split is undertaken along the dimension 1 again. The level number is 3 and the region is the original one added with $d^{l-1}$.

Yeh [YEH90] has proposed BANG file concurrency for the distributed databases systems and multi-tasking operating systems. The most important feature is to increase the throughput. Hosur, Lu and Hedrick [HOS92] proposed dynamic addition and removal of attributes in BANG files to make the BANG file more robust and efficient in different queries.

However, the problem incurred from BANG file is the query. Although the exact match queries in BANG file usually require one pass from root to leaf, it may take a longer traversal for non-uniform data distributions. In light of range queries and partial queries, the BANG file may not be good enough (same as GBD-tree) in view of the nested structure.

## Spatial Indexing

Since the multi-attributes are related to and used by spatial data, the spatial data structures are briefly discussed here. Spatial data have extents (area or volumes). Unlike the point data, the spatial data might frequently overlap. The overlapping of spatial data causes the multi-attribute space indexing to be increasingly complex. There are two modes that are used to treat spatial data. One is called "native space indexing" which can be indexed naturally. The other is called "parameter space indexing" which maps the spatial data to "parameter" space [GAE98].

Native space indexing preserves the locality of the data space. In other words, the object close to the underlying native space will be close to the space as indexed. This method organizes the space objects based on their locations. The hyper-area of nodes encloses all the hyper-area of their children nodes. The R-tree developed by Guttman [GUT84] is one of the preliminary methods for native space indexing. This method extends the B-tree to k-dimensions. It is a height-balanced tree with leaf nodes pointing to the data objects. Each entry in leaf node contains lower and upper points of the rectangle which minimum encloses the data objects, and the entry in non-leaf node contains the upper and lower points of the rectangle that completely encloses the children nodes' area. The data objects may overlap, but they belong to only one of the enclosing rectangles. So, we may search several different paths to find the desired data object. Besides, the splitting of a node is complex. A variant of the R-tree, R*-tree [BEC90] reduces the overlapping area with neighbors to improve search efficiency; however, its splitting method is more complex and inefficient. Searching for data object still requires several searching paths. Berchtold et al. [BER96] proposed a modification of the R*-tree, called X-tree. The X-

tree is designed particularly for indexing high-dimensional data. Another variant of the R-tree is $R^+$-tree [SAM90]. The major difference between the R-tree and the $R^+$-tree is that the non-leaf nodes of an $R^+$-tree's cannot overlap while those of an R-tree's can. The data objects in an $R^+$-tree may appear in several leaf nodes resulting in poor space utilization.

The Quad-tree [SAM90] [BRE93], a hierarchical data structure, which recursively decomposes the space to four equal-sized quadrants. This method is not so good as to deal with the disk I/O access in secondary storage in view of the small (four) fan-outs. The SMR-tree [BAN95] combines the features of both R-trees and $R^+$-trees that decompose the data space to disjoint subspaces. Then, it re-distributes the data objects completely enclosed in the subspaces and moves the data objects partly in both of the subspaces to another SMR-tree. This method improves the search efficiency and space utilization. Another method to improve the search is proposed by Günther and Gaede [GUN97], the oversize shelves, which are attached to the interior nodes of a tree-based spatial access method to avoid the excessive fragmentation in order to have better performance of the search queries.

Alternatively, parameter space indexing transforms a spatial object from a data space to a point in a higher dimensional space, say parameter space. As an example, a data space denoted by the upper and lower points $(x_1, y_1)$, $(x_2, y_2)$ in 2-dimensional space will be transformed to a point $(x_1, y_1, x_2, y_2)$ in a 4-dimensional space. Then, we can apply the multi-attribute indexing methods such as Grid files, BANG files, and G-tree to these higher multidimensional points. As a result, the spatial locality no longer existing is the disadvantage.

# CHAPTER III

# $G^+$-TREE

## Introduction

$G^+$-tree is a data structure associating both grids and B-trees in a specific manner for multi-dimensional data access. It adopts the variable length of a binary numbering scheme to specify grids in the data space. The binary numbers (or called binary codes) are defined as a total ordering sequence; i.e., the grids can be ordered by their binary codes. $G^+$-tree clusters the consecutive grids in a partition and produces the partition number by the smallest grid code. Then, the $G^+$-tree employs a B-tree-like structure to maintain the binary codes (grid codes), and each leaf node of the B-tree-like structure points to a corresponding bucket node which stores the tuples' keys for the corresponding partition.

The main idea of $G^+$-tree is to distribute point data equivalently to two partitions when some partition is overflowed. This idea is different from G-tree which distributes point data simply to fit in two hyper-rectangles. $G^+$-tree is also distinct from GBD-tree which nests in the split block. Nevertheless, to make the point data distributed equivalently into two blocks, the $G^+$-tree's blocks could no longer be hyper-rectangles. However, the equivalent distribution of point data to two blocks in $G^+$-tree can attain at least 50% of the space utilization which is much better than possibly 5% in the worst case

of the G-tree. Therefore, $G^+$-tree can achieve better performance especially in the worst case.

The other benefit of the $G^+$-Tree appears when range query is submitted. Like the G-tree, $G^+$-tree keeps the property of some spatial locality among the point data and the bucket nodes. Meanwhile, the point data in the structure have been tried to be closely stored in the tree if they are close to each other in the data space.

## Data Structure

In this section, the data structure of the G⁺-tree will be described in detail: including the assumptions, the grid numbering scheme, the partition numbering scheme, the definition of the total order for the grid code, and the G⁺-tree algorithms.

### Assumptions

1. The data space is n-dimensional and the dimensions are numbered 0,1,2,3,....,n-1
2. The range of dimension $i$ is $[l_i, h_i)$; i.e., the location of a point on dimension $i$ is $x_i$, and $l_i \le x_i < h_i$.
3. All of the point data (or tuples) lie in this n-dimensional space.
4. The maximum number of keys per non-leaf node is $M$.(The maximum number of descendents per node is $M+1$.)
5. The minimum number of keys per non-leaf node is $m$ except the root node, where $m = \left\lceil \frac{M}{2} \right\rceil$.
6. The maximum number of keys per leaf node (bucket node) is $M_b$.
7. The minimum number of keys per leaf node (bucket node) is $m_b$ except that the total number of the split partitions is less than $m_b$, where $m_b = \left\lceil \frac{M_b}{2} \right\rceil$.

Grid Numbering and Splitting Scheme

This numbering and splitting scheme is adopted from G-tree[KUM94]. A grid is numbered as a binary string of 0's and 1's. Initially, the whole data space is a grid with empty string. Then, the scheme splits a grid into two equal sized grids along one dimension. The lower grid along this dimension is assigned with the original grid string appended with '0'; and, the higher one, with the original string appended with '1'. The following split will be along another dimension on the overflowed grid only. This scheme is to split the grid cyclically along the dimensions. Fig. 4 is an example of the grid numbering for 2-dimension data space. Fig. 4(a) shows that the entire data space is divided evenly by splitting its range along dimension 1 (horizontal coordinate) into two sub-partitions (new grids), numbered 0 and 1 for lower and higher sub-partitions (grids), respectively. Next split, as in Fig. 4(b), when a sub-partition (grid) is split again, proceeds along dimension 2 (vertical coordinate) and the right of denoted number is concatenated with 0 (lower sub-subpartition) or 1 (higher sub-subpartition). The splitting scheme splits the space re-cyclically along the consecutive dimensional coordinate. Fig. 4(c) and (d) shows the results of grid 01 split and 011 split.

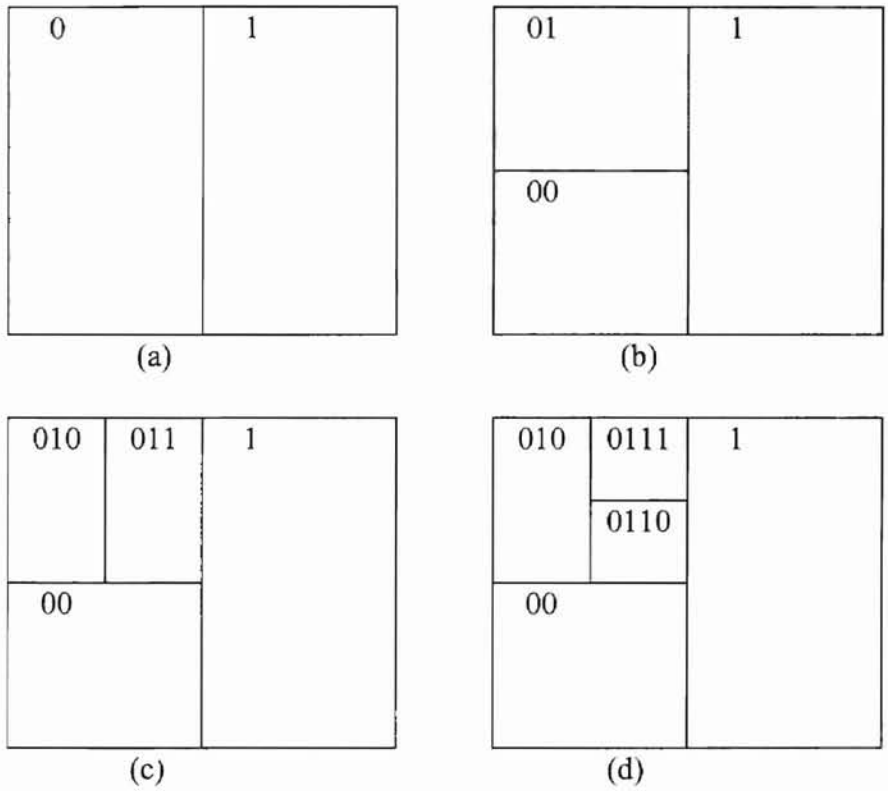


Fig. 4 Grid numbering scheme

(a) Divide the initial data space along the dimension 1, and then assign 0 and 1 to the lower and higher partitions, respectively.

(b) Divide grid 1 along the dimension 2, concatenate 0 and 1 to the new lower and higher partitions to be 00 and 01, respectively.

(c) Divide grid 01 along the dimension 1, concatenate 0 and 1 to the new lower and higher partitions to be 010 and 011, respectively.

(d) Divide grid 011 along the dimension 2, concatenate 0 and 1 to the new lower and higher partitions to be 0110 and 0111, respectively.

Definition for the Total Ordering Scheme

Assume two binary codes denoted by P1 and P2 with b1 and b2 bits long, respectively.

Let b be the length of the shortest one of b1 and b2.

So, b=min(b1, b2). // The length of the shortest one between b1 and b2.

Let MSB(P, b) be a function to extract b from the most significant bits of P (partition number).

1. P1 > P2 if MSB(P1, b) > MSB(P2, b)

Example: P1=011, P2=0101, then,

b=min(b1=3, b2=4) => b = 3

MSB(P1, b) = MSB(011, 3) = 011

MSB(P2, b) = MSB(0101, 3) = 010

⇨ MSB(P1, b) > MSB(P2, b)

⇨ By this rule, P1 > P2

2. P1 < P2 if MSB(P1, b) < MSB(P2, b)

Example: P1=0101, P2=011, then,

b=min(b1=4, b2=3) => b = 3

MSB(P1, b) = MSB(0101, 3) = 010

MSB(P2, b) = MSB(011, 3) = 011

⇨ MSB(P1, b) < MSB(P2, b)

⇨ By this rule, P1 < P2

3. P1 > P2 if MSB(P1, b) = MSB(P2, b) and b1 > b2

Example: P1=0111, P2=011, then,

b=min(b1=4, b2=3) => b = 3

MSB(P1, b) = MSB(0111, 3) = 011

MSB(P2, b) = MSB(011, 3) = 011

⇨ MSB(P1, b) == MSB(P2, b), but, b1 > b2

⇨ By this rule, P1 > P2

4. P1 < P2 if MSB(P1, b) = MSB(P2, b) and b1 < b2

Example: P1=010, P2=0100, then,

b=min(b1=3, b2=4) => b = 3

MSB(P1, b) = MSB(010, 3) = 010

MSB(P2, b) = MSB(0100, 3) = 010

⇨ MSB(P1, b) == MSB(P2, b), but, b1 < b2

⇨ By this rule, P1 < P2

5. P1 = P2 if MSB(P1, b) = MSB(P2, b) and b1 = b2

Example: P1=011, P2=011, then,

b=min(b1=3, b2=3) => b = 3

MSB(P1, b) = MSB(011, 3) = 011

MSB(P2, b) = MSB(011, 3) = 011

⇨ MSB(P1, b) == MSB(P2, b), and, b1 == b2

⇨ By this rule, P1 = P2

Note, definitions 3 and 4 show that binary codes are defined to be greater than the binary codes of their enclosing grids. For example, grid with the binary code, 011, encloses the one with binary code, 0110. But, the order of these 2 binary codes is that binary code 011 > binary code 0110.

Partition Splitting and Numbering

In G-tree, the partition number is also the grid number. That is, a partition is a hyper-rectangle grid. However, in the $G^+$-tree, a partition may consist of several hyper-rectangle grids rather than one. Each partition contains at least $m_b$ point data except in the initial state (the data space initially is the only partition) and at most $M_b$ point data. In order to keep more than $m_b$ of the point data in a partition when it is split, we must redistribute the data points evenly between the two split partitions; i.e., divide the partition into two sub-partitions with no less than $m_b$ data in each sub-partition.

The method splitting an overflowed partition in the $G^+$-trees is to group the consecutive binary grids from the cyclically binary divisions of the partition into two groups. The number of point data in the two groups must be greater than or equal to $m_b$. In the first group, all of the grid codes are less than any of the grid codes in the second group. The partition number is then assigned with the smallest binary code of the grids inside of this partition. Fig. 5 shows the procedure of splitting the initial partition for $M_b$=4, $m_b$=2. Fig. 5(a) splits the initial partition into two grids which codes are assigned with 0 and 1. In the grid 1, there is only one point data. (The number of point data in grid 1 is less than $m_b$; an imbalanced distribution) We, therefore, need to split the other grid again. Fig. 5(b) shows three grids after dividing grid 0 into two grids. In the meantime, there exists three grids with the grid codes 00, 01, and 1. Obviously, we can group grid 00 as lower partition; and grid 1 as higher partition. But, the distribution will not be balanced no matter how these grids are combined as two groups. In other words, grid 01 grouped to either lower (grid 00) or higher (grid 1) partition can not make both of the groups with the number of data no less than $m_b$. Therefore, we have to divide grid 01 into two partitions

to distribute the point data to the lower and higher partitions. Fig. 5(c) is newly resulted from Fig. 5(b). There are four grids; i.e., grid 00 with no data, grid 010 with one data, grid 011 with three data, and grid 1 with one data. Again, in Fig. 5(c), we still can not group the grids as two partitions with both of the number of data no less than $m_b$. But, we can assign grid 00 and grid 010 to the lower partition and grid 1 to the higher partition. Hence, the only unassigned grid, 011, has to be split again. Fig. 5(d) shows the result of the final partitioning scheme: after dividing grid 011, the data point distributed to two new grids, 0110 and 0111, can be distributed to lower partition and higher partition, respectively. Meanwhile, we obtain the balanced distributed data points in the lower and higher partitions. The constituent in the lower partition are grids 00, 010, 0110; in the higher partition are grids 0111 and 1. Finally, the partition numbers (codes) can be determined. In this example, 00 is determined in the lower partition and 0111 in the higher partition. Both are the smallest grid codes in the order sequence among the constituents.

Fig.6 shows the other example of splitting non-rectangle partition (following Fig. 5). Fig. 6(a) shows the first division of the partition 0111 which includes grids 0111 and 1. The lower partition, grid 0111 does not balance with the higher partition, gird 1. Fig. 6(b) continues to divide the higher partition, grid 1, into grids 11 and 10. Nevertheless, the lower partition contains only grid 0111; the higher partition contains grid 11. Grid 10 is going to be divided to distribute its data points to the two partitions. Fig. 6(c) shows the division of grid 10. After grid 10 being divided, the grid 100 can be placed in the lower partition and the grid 101 in the higher partition. In view of the definition, the partition number is the smallest number among the constituent grids. Thus, the lower partition is

assigned with 0111 and the higher partition with 101. The result is shown in Fig. 6(d). Each grid is divided cyclically along every dimension. Note, the grids 0111 and 100 belong to the same partition, 0111.

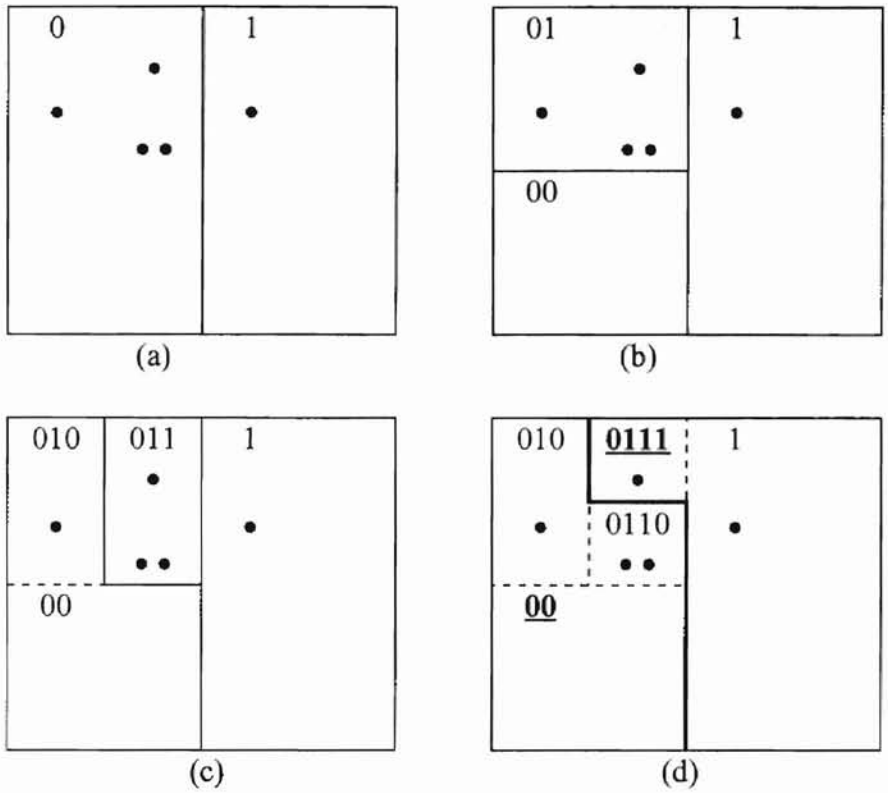


Fig. 5 Splitting and partitioning scheme for overflow in the bucket node of $G^+$-tree ($m_b$=2 and $M_b$=4).

(a) Divide the space along dimension 1 (horizontal coordinate). The distribution is not balanced for the lower and higher partitions are not balanced.

(b) Divide grid 0 along dimension 2 (vertical coordinate), still imbalanced.

(c) Divide grid 01 along dimension 1, still imbalanced in two partitions for any combinations of lower grids and higher grids.

(d) Divide grid 011 along dimension 2 into two grids. Lower partition with grids 00, 010, and 0110 has $m_b$ +1 data points. Higher partition with grids 0111 and 1 has $m_b$ data points; i.e., balanced. Then, the partition number is assigned to each partition by choosing the smallest constituent grid numbers, 00 and 0111, respectively.

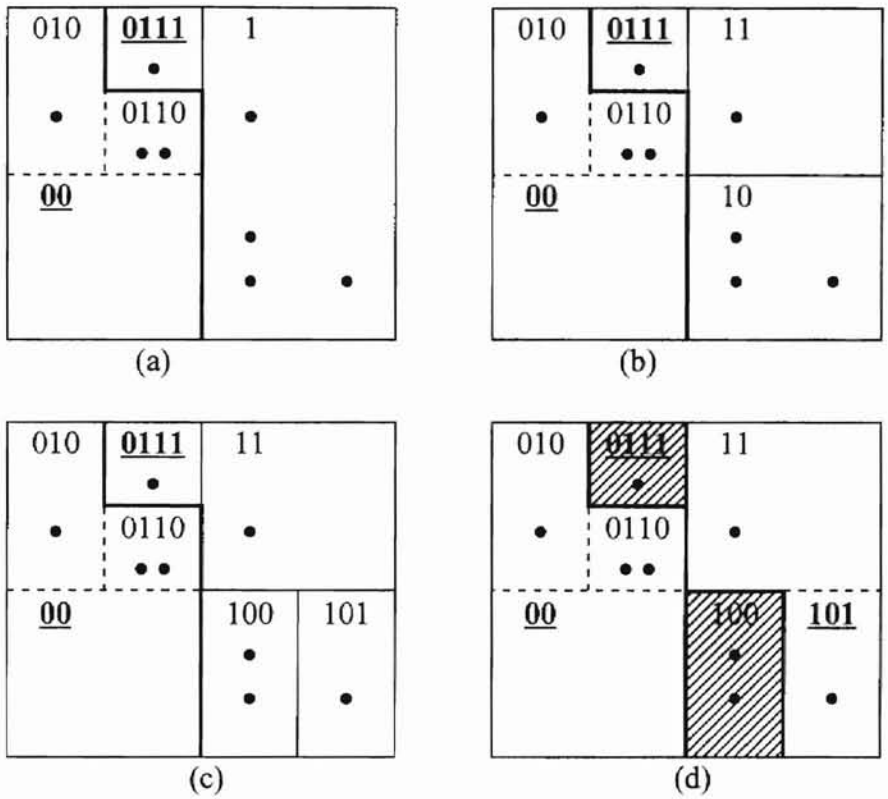


Fig. 6 Splitting and partitioning scheme for inserting two more data points into partition 0111 of Fig. 5(d)

(a) Divide along dimension 1 (horizontal coordinate), lower and higher partitions (grid 0111 and grid 1) are imbalanced.

(b) Divide grid 1 along dimension 2 (horizontal coordinate), imbalanced. The lower partition contains grid 0111; higher partition contains grid 11.

(c) Divide grid 10 along dimension 1 into two grids. The lower partition with grids 0111 and 100 has $m_b$ +1 data points; the higher partition with grids 101 and 11 has $m_b$ data points; i.e., balanced. Then, the partition number is assigned to each partition by choosing the smallest constituent grid number, 0111 and 101, respectively.

(d) This is the result. Note that grids 0111 and 100 belong to the same partition 0111, the same bucket node.

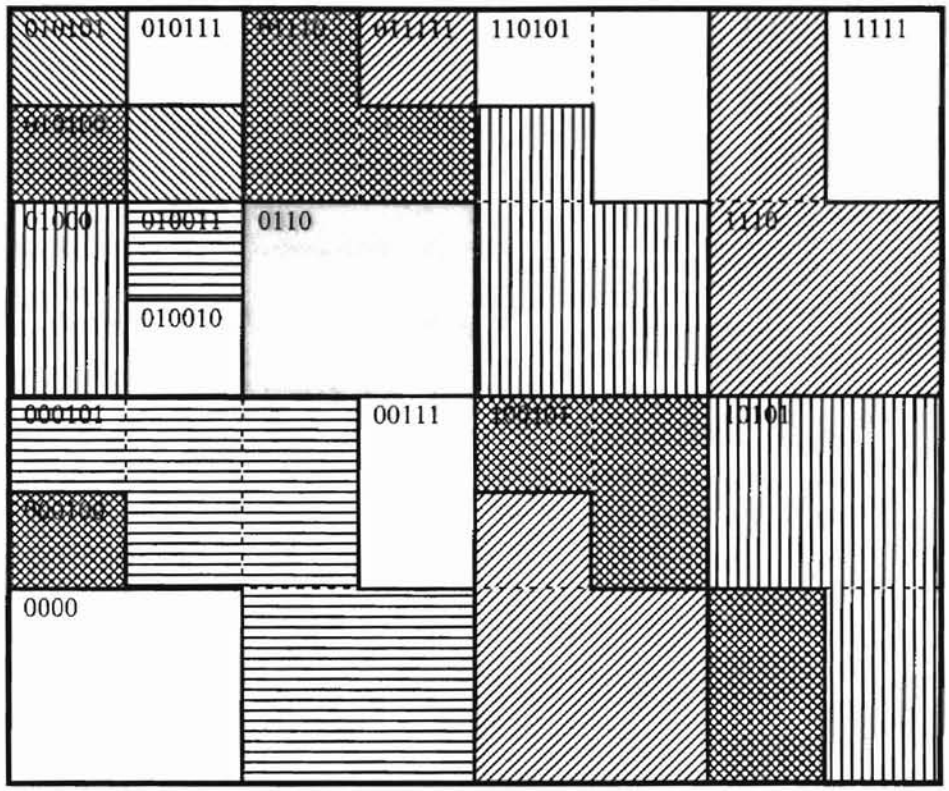


Fig. 7 A possible example of $G^+$-tree for two dimensional space

The partitions of $G^+$-tree could be hyper-rectangles or non-hyper-rectangles.

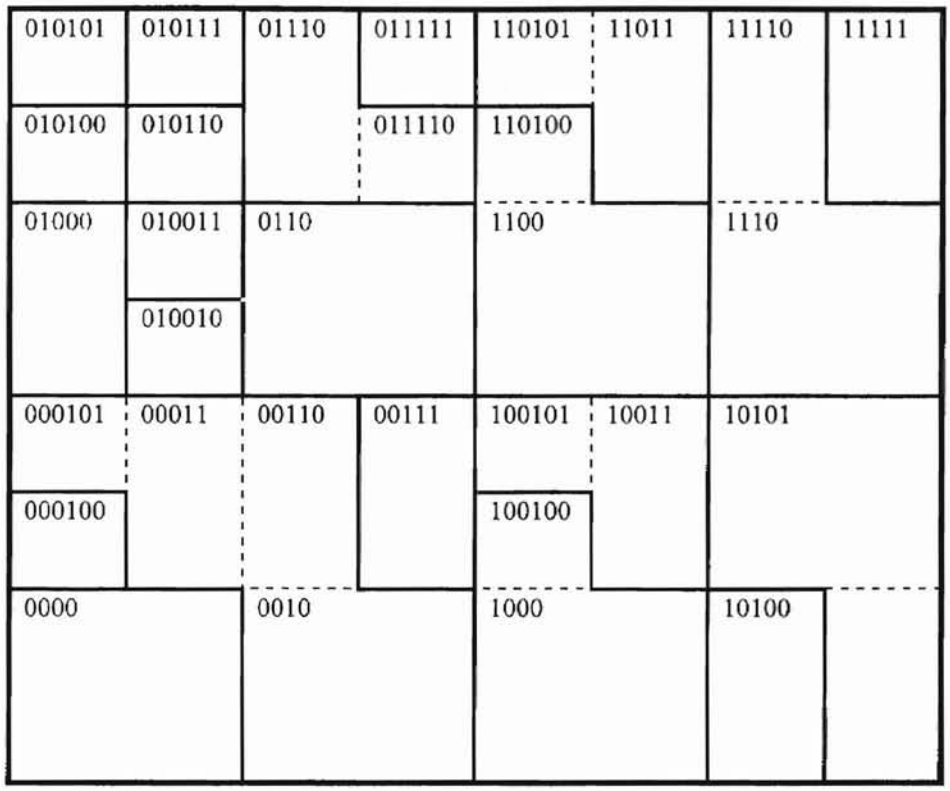


Fig. 8 The constituent grids for the example in Fig.7

Note that most of the partitions keep the property of locality even though they are not hyper-rectangles. The others keep the two groups of data with property of locality.

Examples:

Partition 0000 has only one grid, 0000.

Partition 000101 has constituents of grids 000101, 00011, 0010, and 00110. It is not a hyper-rectangle.

Partition 011111 has constituents of grids 011111, 1000, and 100100. This partition has two groups of data. Each group keeps the property of locality within itself. But, the two groups are far away from each other.

The Main Features

1. The data space is decomposed into non-overlapping partitions in variable size.
2. All the partitions together can span the whole data space.
3. Each partition corresponds to a disk page (or bucket).
4. The number of entries in a partition is between the minimum number of entries ($m_b$) and the maximum ($M_b$) except that only one partition exists in the data space (the total number of entries is less than or equal to $M_b$).
5. The partitions keep the property of locality, i.e., the entries inside the partition are close to each other. In the worst case, two groups of data inside a partition are far from each other. But, the data are close to each other within the same group.
6. A partition consists of the consecutive grids that are binarily split from the data space cyclically along each dimension to meet the above requirements.
7. Each partition is assigned a unique partition code that is the least grid code of the constituted grids.
8. The total ordering is defined over the partition codes (grid codes) so they can be stored in a $B^+$ tree with the order of M. That is, the maximum number of keys and pointers in a non-leaf node is M and M+1, respectively.

## Algorithms

From last section's description, a short definition for this data structure can be made as follows:

1. $G^+$-tree is a height-balanced tree like $B^+$ tree.
2. The sub-tree of $G^+$-tree is also a $G^+$-tree.
3. There are two types of nodes in a $G^+$-tree:
   (1) Index node (non-leaf node), which contains M keys (partition numbers) and M+1 pointers to the child nodes.
   (2) Bucket node, which contains all the entries corresponding to its partition.
4. Inserting one more point datum into a full partition makes the partition split into two sub-partitions. Each of which contains no less than $m_b$ entries. (i.e., the number of point data $\geq m_b$).
5. Deleting one point datum from a partition which contains only $m_b$ point data makes its point data redistributed or merged to neighbor partitions.

There are two types of nodes in the $G^+$-tree index structure. One is the internal node which is the same as that of $B^+$-tree's or G-tree's. The maximum number of keys in an internal node is M. Also, the minimum number of keys in an internal node except the root node is m which is less than or equal to the ceiling of M/2. The minimum number of keys in root node is one. The keys in a node are ordered according to the rule of total ordering of grid codes. A node with k keys has k+1 children in the next level. All the keys in the child i are greater than or equal to key i-1 but less than key i.

The other type of node is leaf node also named bucket node. This node contains all the coordinates of each data and the addresses to the indexed file. The maximum number

of the point data in a node is $M_b$ and the minimum number of keys is $m_b$ which is less than the ceiling of $M_b/2$.

Before any operation executed, the current longest grid code (smallest in partition size) for the requested data has to be determined in order to locate the enclosing partition. Then, search the $G^+$-tree by descending from the root in the manner similar to a B-tree to find the actual partition number which corresponding partition encloses the grid (partition number $\leq$ requested grid number $<$ next partition number). Thus, we need to keep a global variable, b, for the current longest length of the grid code.

$G^+$-Tree Algorithms assume:

1. There are n dimensions in the data space.
2. b is the length of the grid number which is the longest in the system.
3. MAX_VALUE is an array to keep the maximum value for each coordinate.
4. MIN_VALUE is an array to keep the minimum value for each coordinate.

The algorithms are all similar to those of $B^+$-tree since $G^+$-tree takes the $B^+$-tree as the index tree to the partition with partition numbers as keys.

Grid Determined for a Point Data

As all the partitions are variable in shape (might not be a hyper-rectangle), to locate a point datum we have to locate the enclosing grid first. In other words, we have to find the current smallest grid that encloses the point data. Then, we can use this grid code to locate the partition that encloses this grid (and the point data) in the $G^+$-tree and find the corresponding tuple. The Grid_Determine function is served for this purpose.

---

---

Grid_Determine(P, T)

Input:

P: the coordinates of the point data

T: the $G^+$-tree

b: the longest bit number for a grid so far. (b is a global variable)

Output:

G: the grid code of the current smallest sized grid that contains P.

---

i: index variable

```
1. {
2.   G="";                    // initialize the grid number as an empty string
3.   for (i = 1; i ≤ n; i++)   // assign the max and min value for each dimension
4.   {
5.       l_i = MIN_VAL[i];    // minimum value along the dimension i
```

```
6.      h_i = MAX_VAL[i];    // maximum value along the dimension i
7.  }
8.  for (k = 1; k ≤ b; k++)          // repeat b times since the current longest length of a
                                              grid code is b
9.  {
10.     i = ((k-1) mod n ) + 1;      // decide which coordinate to be split along

11.     // binary split along dimension i
12.     if ( x_i < (l_i +h_i )/2 )   // the data located in the lower grid
13.     {
14.         concatenate '0' to right of G ;    // New Grid code
15.         h_i = (l_i +h_i )/2 ;    // the higher boundary moved to the new position
16.     }
17.     else                         // the data located at the higher grid
18.     {
19.         concatenate '1' to right of G;     // New Grid code
20.         l_i = (l_i +h_i )/2 ;    // the lower boundary moved to the new position
21.     }
22. }
23. return (G);                      // The current longest grid code enclosing data
24. }
```

The Grid_Determine is to determine the grid which encloses the point data. The grid code is b bits long ( the current smallest size of a grid). First of all, the maximum and minimum values of each dimension are assigned to low and high for each dimension (on line 3 to line 7). Then, like the binary search, narrow down the low and high point cyclically along each dimension until the length of grid code is b bits long(on line 8 to line 22). The grid number is appended with '0' when data is inside the lower part (on line 12 to line 16); appended with '1' when it is inside the higher part. After b times of narrowing down (on line 17 to line 21), the grid is found (the grid number is b bits long).

Algorithm Search

Given a $G^+$-tree whose root node is T and a point data $P=(x_1, x_2,..., x_n)$, find the tuple for the point data $P=(x_1, x_2,..., x_n)$.

---

---

Search(P, T) //P: point data; T: $G^+$-tree

Input:

P: the point data to be looked for

T: the $G^+$-Tree

Output:

Address: the tuple's address in the indexed file if found; -1 if not found.

---

G: the grid number which accommodates the point data.

$E_i$: the entry to the next node of $G^+$-tree.

```
1. {
2.   G = Grid_Determine(P, T);
3.   while (T is not a leaf node)
4.   {
5.       /* find the entry to the next level */
6.       Let T = entry E_i        if KEY_i ≤ G < KEY_{i+1}
7.           or T = E_{M+1}       if KEY_M ≤ G.   // M: the max number of keys in the node
8.   }
9.   /* Now, T is a leaf node */
```

10. Check the entries in the leaf node to determine whether there is a key same as P which is $(x_1,x_2,...,x_n)$.
11. If the point data found in the leaf node, return address of the tuple in the indexed file
12. else return NOTFOUND.
13.}

---

Description:

To search a point datum, we invoke the Grid_determine to locate the grid which encloses the point data (on line 2). Then, use this grid number (G) as a key to traverse down to the $G^+$-tree to the leaf node (on line 3 to line 8). At the leaf node, look for the entry which is $(x_1, x_2, \ldots, x_n)$ (on line 10). It will return the address of the tuple location in the indexed file if it is found (on line 11); NOTFOUND (-1) ; otherwise (on line 12).

Algorithm Insertion

Given a $G^+$-tree whose root node is T and a tuple with indexing data $P=(x_1, x_2,..., x_n)$, insert the tuple with the point data $P=(x_1, x_2,..., x_n)$ to T.

Inserting a new point data is similar to the insertion in B-tree. That is, data are added into the leaf node. If that node overflows, it must be split and insert the new node key to the parent. If parent node overflows after the insertion, the parent node has to be split again. That is, the splits may propagate up to the tree root.

---

---

Insertion(P, T)

Input:

P: the point datum which is to be inserted to the $G^+$-tree

T: the $G^+$-tree to.

Output:

T: the new $G^+$-tree which encloses the new point datum P.

---

```
1. {
2.  G = Grid_Determine(P);  // To locate the grid no. G for operations in the G+-tree.
3.  TreeInsert(P, G, T);    // To insert key G to the G+-tree
4.  if (overflowed)         // if the G+-tree's root split, create a new root.
5.      Create a new node as the root to accommodate the 2 nodes that split from the old
                            root;
6. }
```

---

---

TreeInsert(P, G, T)

Input:

P: the point data to be inserted into the tree

G: the grid no. to accomodate the point data P.

T: the $G^+$-tree

Output:

T: the (sub)$G^+$-tree with P inserted

PromotedKey: the key promoted if the node split

PromotedAddr: the address of the promoted node if the original node split

---

```
1.{
2.  if (T is not a leaf)
3.  {
4.      /* find the entry to the next level */
5.      Find the entry E_i if KEY_i ≤ G < KEY_i+1 or E_M+1 if KEY_M ≤ G.
6.      TreeInsert(P, G, E_i );              // recursively down to leaf node
7.      if (propagating up for insertion)
8.      {
9.          add the new entry to the node;
10.         if (overflowed)
```

```
11.          index_split(T);              // non-leaf node split
12.     }
13. }
14. else                                  // T is a leaf node
15. {
16.     Check the entries in the leaf node to determine whether there is a key same as P
          which is (x1,x2,...,xn).
17.     if exist, return error of data collision exist.
18.     else add P to this node
19.     if (overflowed)
20.          bucket_split(T);             // split the bucket node if overflowed
21. }
22.}
```

---

Description:

To insert a data, several steps are required:

1. Find the current smallest size grid (longest in grid code length) to which the point belongs by calling the function of Grid_determine (on line3 of Insertion).
2. Search for the actual partition to which the point belongs in the $G^+$-tree by invoking the recursive Insert function, TreeInsert, to reach the leaf node.
3. At the level of leaf node, if the partition found has one more vacancy, the TreeInsert function just puts the point data into it and, then, stops. Otherwise, split the partition into two sub-partitions by redistributing the $M_b$ +1 data according to the partition

number scheme described and return the new partition number to the upper level ($G^+$-tree) for propagating up (by invoking the function of bucket_split).

4. At the level of non-leaf node, if the node has one more vacancy, the TreeInsert just puts the point data into it and, then, stop. Otherwise, split the node into two by redistributing the M +1 point data and return the new partition number(s) to the upper level ($G^+$-tree) for propagating up (by invoking the index_split).
5. If it overflowed at root node, then, create a new node. Distribute those point data including the promoted key evenly distribute between the old root and the new node. Then, create another new node for indexing these 2 redistributed nodes. Therefore, this finally created node becomes the new root node.

Algorithm Bucket Split

Given a datum P, its corresponding grid G, and a full leaf node T, this function creates a new node and evenly distributes data between the old and the new node, and then return them to the parent.

---

---

Bucket_Split (P, G, T) // P is added to T, G is the grid code, T is a full node.

Input:

P: the point data to be inserted into the full node T

G: the grid to accommodate the point data P.

T: the full leaf node in $G^+$-tree

Output:

T: the (sub-)$G^+$-tree with P inserted

PromotedKey: the key promoted after split

PromotedAddr: the address of the promoted node after

---

```
1.{
2.  for (k = 1; k ≤ n; k++)      // assign the max and min value for each dimension
3.  {
4.      l_i = MIN_VAL[i];        // minimum value along the dimension i
5.      h_i = MAX_VAL[i];        // maximum value along the dimension i
6.  }
```

```
7. Create a new bucket node, T';                    // create a new node
8. Create 3 new temporary nodes, named workingNode, tmp1, tmp2 which are able to
                accommodates M_b+1 keys.
9. Move all the entries of T and the new point data P to workingNode which has, then,
                M_b+1 keys. (So, T is empty now.)
10. done = false;
11. G' = "                          // initial the grid number of T to empty
12. k = 0                           // number of split
13. while (not done)                // entry number of node < m_b
14. {
15.     k ++;                        // the kth time for the redistribution
16.     i = ((k-1) mod n ) + 1;      // decide which coordinate to be split along
17.     Move entries with x_i < (l_i +h_i )/2 ) in workingNode to tmp1  // the data located at
                                                                        // the lower grid
18.     Move entries with x_i >= (l_i +h_i )/2 ) in workingNode to tmp2 // the data located at
                                                                        // the higher grid
19.     if ((EntryOf(tmp1)+EntryOf(T) >= m_b) and
                (EntryOf(tmp2)+EntryOf(T') >= m_b))          // done
        {
20.         Move all the entries of tmp1 to T;
21.         Move all the entries of tmp2 to T' ;
22.         b = the longest bit length of G and G';
23.         if (MSB(G', b) == G)            // the last split occurs at the smallest
```

```
                                                    // grid of partition G
24.         {
25.             G = G' concatenated with '0';       // the smallest grid code changed
26.         }
27.         G' = G' appended with '1';              // for the new grid
28.         done = true;
29.     }
30.     else if (EntryOf(tmp1)+EntryOf(T) < m_b )
31.     {
32.         Move all the entries of tmp1 to T.
33.         Move all the entries of tmp2 to workingNode   // entries of tmp2 need to be
                                                          redistributed
34.         concatenate '1' to the left of G' ;           // the grid to be re-split is at
                                                          higher part
35.     }
36.     else        // EntryOf(tmp2)+EntryOf(T') < m_b
37.     {
38.         Move all the entries of tmp2 to T';
39.         Move all the entries of tmp1 to workingNode;  // entries of tmp1 need to be
                                                          redistributed
40.         Concatenate '0' to G' ;                       // the grid to be re-split is lower one
41.     }
42. }       // end of while (not done)
```

43.}

---

Description:

The strategy of splitting a partition is to divide the original partition several times according to the method we used in partition numbering and splitting until both of the sub-partitions have the evenly distributed point data. In other words, we divide the partition into two sub-partitions and distribute the point data into these two sub-partitions. It is accomplished if all the point data are evenly distributed; otherwise, the sub-partition with more than $m_b$ point data needs to be divided again along the next dimension. It is like to divide the partition into small grids, then group the grids as two partitions which have the evenly distributed number of point data.

Line 2 to line 5 of this algorithm set up the highest and lowest of each dimension for later use. Line 7 creates a new node to keep the data in the higher part of the overflowed partition. Line 8 creates three temporary nodes, workNode, temp1, and temp2. The workNode keeps all the data which have not been assigned to any of the two partitions yet. The temp1 (temp2) keeps the data on the lower part (higher part) when division occurs along some dimension.

On line 13 to line 44, it cyclically divides the space along each dimension to separate the data into two partitions.

Algorithm Index Node Split

Given a full internal node (index node) T, a promoted node (which key is PromotedKey and address is PromotedAddr), this algorithm splits the full internal node T evenly into two nodes while the promoted node is inserted, and then returns them to the T's parent node.

---

---

Index_Split(T, PromotedKey, PromotedAddr)

Input:

T: the full non-leaf node in $G^+$-tree with M keys (M+1 entries to children)

PromotedKey: the key promoted from the child level

PromotedAddr: the address of the promoted node

Output:

T': the new sub-$G^+$-tree which is separated from the original full node T.

PromotedKey: the key promoted after split

PromotedAddr: the address of the promoted node after split

---

```
1.{
2.      CreateNewNode(T');          // create a new node
3.      Distribute the keys (M+1) and children (M+2) in T to T and T' evenly with
                    smaller half to T, the larger half to T'.
4.      The middle key and the T' node address are promoted (to parent level).
```

5.}

Description:

The strategy of splitting the index node is the same as that of $B^+$-tree. The keys in the overflowed node are in linear order. The index node has M+1 keys and M+2 entries (overflowed) to children. It redistributed the first (M+1)/2 and last (M+1)/2 to two nodes (the node T and a new node T'). Hence, the middle of the keys is the promoted key to the upper level of $G^+$-tree; and the address of the second node (T') is the promoted address. Fig.9 is the procedure to split the full index node.

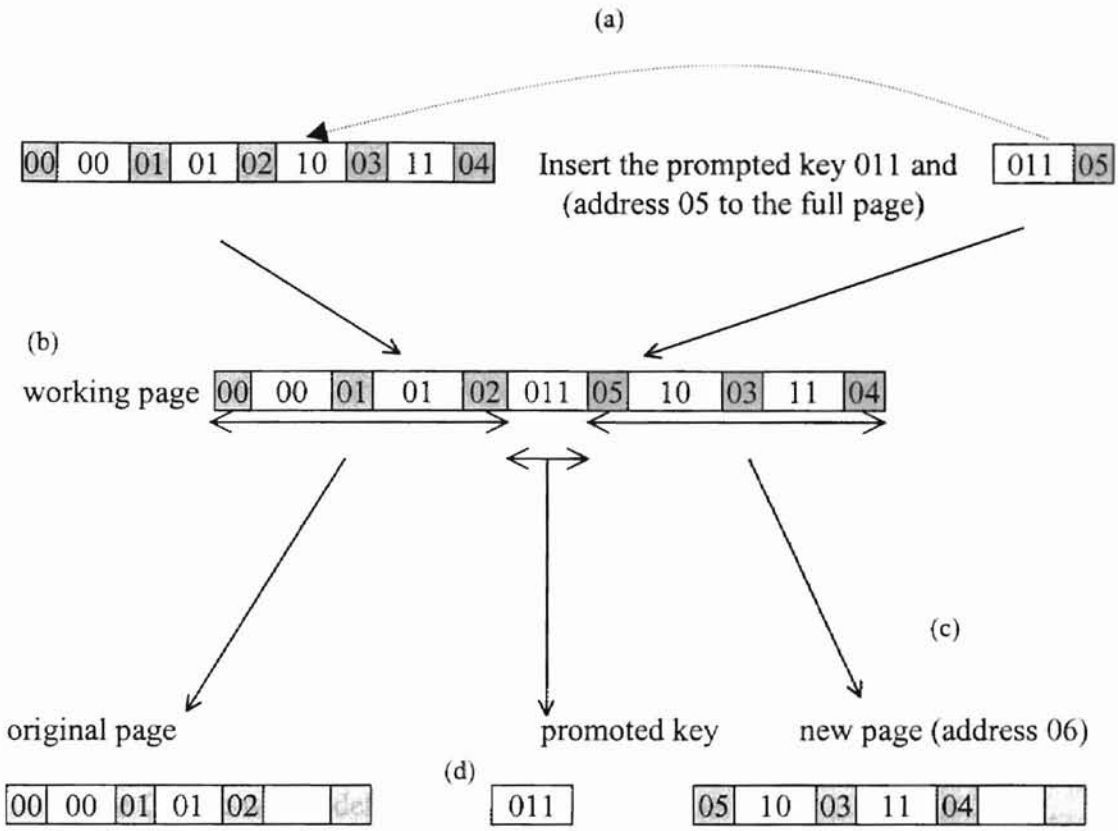


Fig. 9 The steps of node split

In this figure, white box is the key; black box is the pointer to the child node. The maximum number of keys in a node is 4; the minimum is 2.

(a) Initially, there are keys 00, 01, 10, and 11 in a full node. Then, a key 011 is promoted from the child node.

(b) Create a working node to keep all of the keys, and then redistribute those data into two groups.

(c) Create a new node and let the old node keep the lower keys, 00 and 01; and new node keep the higher keys, 10,11.

(d) Promote the middle key 011 to the parent with the new node address (06).

Algorithm Deletion

Given The $G^+$-tree T and the datum P which is going to be deleted from T, then return the new $G^+$-tree with P deleted.

---

---

Deletion(P, T)

Input:

P: the point data to be deleted

T: the $G^+$-tree

Output:

T: the $G^+$-tree with P deleted

G: the smallest sized possible grid no. that the point datum P lies on.

---

```
1.{
2.  G = Grid_Determine(P);
3.  DeleteInIndexnode(P, G, T);
4.}
```

---

Description:

Deletion algorithm includes three steps:

1. Determine the current smallest grid that contains the point data by invoking Grid_Determine.
2. Search the grid in the $G^+$-tree and find the point data in the found node.

3. Delete the point datum in the bucket node if it exists.

   3a. Done if the number of data left in this node is greater than $m_b$.

   3b. Redistribute the data with the sibling (left or right) if the number of data in bucket node is less than $m_b$ and the number of data in the sibling is greater than $m_b$.

   3c. Move the keys to the sibling and delete itself if the number of data in this bucket node is less than $m_b$ and the number of data in the sibling is less than $m_b$. Then, propagate the deletion to the upper level of the $G^+$-tree.

In this algorithm, line 2 is for step 1, line 3 for step 2, and step 3 by calling the other function (DeleteInIndexnode) which is listed in the next page.

---

---

DeleteInIndexnode(P, G, T)

Input:

P: the point data to be deleted from $G^+$-Tree, T

G: the grid number to accommodate the point data P.

T: Node of the $G^+$-tree

Output:

T: the (sub)$G^+$-tree with P deleted.

PropagatedKey: the key propagated for deletion.

---

1.{

```
2.  If T is leaf node                          // for leaf node
3.  {
4.      DeleteInBucketnode(P, T);
5.      if redistribution after delete P
6.          return PropagatedKeyOfRedistribution
7.      else if deletion propagating
8.          return DeletionPropagation, PropagatedKey.
9.      else
10.         return NoDeletionPropagation.
11. }
12.  // T is non-leaf node
13. In the node T,  find the entry E_i to the node in the next level for P
14. DeleteInIndexnode(P, G, E_i);      // recursively down to the leaf node
15. if redistribution in the lower level
16. {
17.     replace the key with the propagted key
18.     return NOPROPAGATION
19. }
20. if no deletion propagated from the lower level
21.     return NOPROPAGATION
22. // deletion propagated from the next level
23. if the number of the keys in the node > m          // m is the minimum no. in a node
24. {
```

```
25.     delete the key and compact the node
26.     return NOPROPAGATION
27. }
28. if the left sibling exist and its number of keys > m        // Need sibling's
redistribution
29. {
30.     move the key of parent and left sibling's right-most child to this node
31.     return propagated key of right-most key of left sibling
32. }
33. if the right sibling exist and its number of keys > m
34. {
35.     move the key of parent and right sibling's left-most child to this node
36.     return propagated the key of the most left key of the right sibling
37. }
38. // No redistribution possible, concatenate with the parent key and the sibling
39. Combine the key and left sibling node and this node together to form a new node.
40. return deletion propagation
41.}
```

---

Description:

This is a recursive function of deleting a point data in the $G^+$-tree. First, go from top of the $G^+$-tree down to the leaf node and delete the point datum at the bucket node (which is listed next). At each level, there are four conditions that we have to take into

consideration to delete a data (key). (For convenience, we can let the number of keys of its left sibling be 0 if a node has no left sibling.)

1. The number of keys in a node is still greater than or equal to m.

   Then, just return.

2. The number of the keys in a node is less than m, but that of its left sibling's is greater than m.

   Then, get one key from the left sibling and change the parent's key pointer to this node.

3. The number of the keys in a node is less than m and that of its left sibling's is equal to m; but, that of the right sibling's is greater than m.

   Then, get one key from the right sibling and change the parent's key pointer to the right sibling.

4. The number of the keys in a node is less than m and those of both of its siblings' are equal to m.

   Then, merge with the left sibling if exists; merge with right sibling, otherwise

However, if the current node is a root node, then, we do not consider the situation for the number of keys less than m.

Delete a datum in a $G^+$-tree at the bucket node:

Given a datum P, a $G^+$-tree T (bucket node level), then delete the data P in $G^+$-tree T if exists one. This function is called by DeleteInIndexnode which is listed at page 56.

---

---

DeleteInBucketnode(P, T)

Input:

P: the point data to be deleted from $G^+$-Tree, T

T: Node of the $G^+$-tree

Output:

T: the (sub)$G^+$-tree with P deleted.

PropagatedKey: the key propagated for deletion.

---

```
1.{
2.  If P exists in the node T
3.      delete it
4.  else
5.      return error no such point data
6.  if current node is the only one node in the G+-tree        // only node in the G+-tree
7.  {
8.      if no data in the node                                 // empty after delete P
9.      {
10.         delete the current node
```

11. return the propagated key // return to the root for deleting the key

12. }

13. else

14. return NoPropagated

15. }

16. if the number of keys >= $m_b$ //$m_b$ is the minimum number for a bucket node

17. return NOPROPAGATION

18. // The following code block means the number of keys in the node < $m_b$

19. if the left sibling exists and the number of keys > $m_b$

20. {

21. Redistribute the keys in left sibling and itself by the partition numbering method to reorganize these two partitions and propagated the new current partition number to the parent.

22. return the propagated number

23. }

24. if the right sibling exist and the number of keys > $m_b$ // left sibing's <= $m_b$

25. {

26. Redistribute the keys in its right sibling and itself by the partition numbering method to reorganize these two partitions and propagated the new right sibling's partition number to the parent.

27. return the propagated number

28. }

29. // the number of keys in siblings are ≤ $m_b$

30. Combine with one of the sibling to be one node, and delete the original node.

31. return deletion propagated

32.}

---

Description:

This is similar to the algorithm DeleteInIndexnode(P, G, T). The difference is that it is for a bucket node which has distinct node structure and different maximum and minimum numbers of keys in a node. Besides, the algorithm is not a recursive function call.

A special case for this deletion algorithm: if the current node is the only bucket node in the $G^+$-tree. After the deletion of the data, if the number of data in this only one node is 0, then delete the node; otherwise, just return without considering requirement of the minimum number of keys in a node. (On line 6 to line 15.)

Algorithm Range Query

Given a pair of points to be the range query's hyper-rectangle to get all the point data from the $G^+$-tree, G. (The two points are the lower left and right upper points of that hyper-rectangle.)

---

---

Input:

R: the range of query ($x_1$ , $x_2$ , ..., $x_n$, $y_1$, $y_2$, ..., $y_n$)

T: the $G^+$-tree

Output:

$P_{return}$: the linked list of the point data

---

B: the bucket node in the $G^+$-tree

$B_{next}$: the right neighbor of B in the $G^+$-tree T.

1.{

2. Divide the query range into a set of smallest sized grids and, then, order these grids according to their grids codes into a linked list L.

3. $P_{return}$ initialized to be empty.

4. while (L is not NULL)

5. {

6. Choose the first grid, G, in the linked list L as a key to search in the $G^+$-tree to get the bucket node B.

7. Put B's point data, which locate inside the range query, to $P_{return}$.

8. // Delete the grids, which are included by B, from L.

9. Delete the grids with grid codes < $B_{next}$ in L

10. }

11. return $P_{return}$. // return all the data meet the range query.

12.}

---

Description:

The range query is specified by a hyper-rectangle which is indicated by two points. To get the data for this range query, first of all, is to find those partitions which have overlapped (even slightly) with this hyper-rectangle. Hence, the range query hyper-rectangle is cut into the current smallest sized grids for searching the $G^+$-tree to locate these partitions (on line 2). After then, use these grids to find the partitions (the bucket nodes). As part of the point data in a located partition might not be inside the range query, they are filtered to get the desired points (on line 7). Since several grids of L's may be in a located partition, it is not necessarily searched down for the same partition. Thus, the grid within this located partition can be deleted (on line 9). Finally, after searching for those grids, return the point data found in the range.

# CHAPTER IV

## PERFORMANCE ANALYSIS

In this section, the performance is analyzed and compared to the G-tree. As very large databases are assumed, the data are stored in a permanent memory, the disk. Therefore, the performance for the data retrieving operations is bound by the disk I/O. The data retrieving operations include insertion, search, deletion, and range query.

For normal data distribution, $G^+$-tree and G-tree structures are simulated and tested to measure their storage utilization. The storage utilization for bucket nodes is $\frac{\text{(the number of data in the data space)}}{\text{(the number of bucket nodes used)} \times \text{(the number of entries per node)}}$. The experiments were run with a large number of randomly generated data on Unix system and repeated several times. The result of the storage utilization for $G^+$-tree is 67±5% and that of G-tree is 69±5%. The result shows that the storage utilization of $G^+$-tree is slightly worse than that of G-tree in the situation of normal data distribution. This means that it is only a little price to pay for $G^+$-tree to eliminate the possible degeneration for G-tree.

For some skew data distribution, the analysis is done by the following description. Assume that there are N point data and the maximum capacity of an internal node is M and minimum capacity is m. Meanwhile, assume that the maximum capacity of a leaf node is $M_b$ and the minimum capacity is $m_b$. Furthermore, in the worst case, every leaf

node has only $m_b$ point data. Hence, the total number of bucket node is $\left\lfloor \frac{N}{m_b} \right\rfloor$.

Use these bucket nodes to build the internal tree ($B^+$-tree); therefore, in the worst case for the internal $G^+$-tree, the number of nodes at the lowest level of internal node is $\left\lfloor \left( \left\lfloor \frac{N}{m_b} \right\rfloor \right) / m \right\rfloor$ and the total number of the internal nodes is $2 \left\lfloor \left( \left\lfloor \frac{N}{m_b} \right\rfloor \right) / m \right\rfloor - 1$. The internal tree height is $\log_m \left\lfloor \frac{N}{m_b} \right\rfloor$ and the overall height of the $G^+$-tree is $1 + \log_m \left\lfloor \frac{N}{m_b} \right\rfloor$. (1 is the level of bucket node.)

From the above description, it is easy to find the worst case for the space complexity in $G^+$-tree and the number of internal nodes plus the number of bucket nodes,

i.e., $2 \left( \left\lfloor \frac{N}{m_b} \right\rfloor \quad m \right) - 1 + \left\lfloor \frac{N}{m_b} \right\rfloor$.

Considering the space utilization, the worst case occurs when the numbers of point data in all bucket nodes are the same as the minimum capacity, $m_b$; i.e., the space utilization in the worst case is $m_b/M_b$. (We do not consider the situation of only one bucket node.) For example, if $m_b$ is half of the $M_b$, the space utilization is 50%.

The time complexity is counted as the number of disk I/O. Searching one point datum, it goes from the root of $G^+$-tree down to the bucket node. Thus, the number of disk I/O is the height of the $G^+$-tree, $1 + \log_m \left\lfloor \frac{N}{m_b} \right\rfloor$.

Insertion has to go from the root of the $G^+$-tree down to the node for the point datum to insert. Prior to a datum insertion, it has taken $(1 + \log_m \left\lfloor \frac{N}{m_b} \right\rfloor)$ times of I/O. The worst

case is to insert the point datum into a full node with all full ancestor nodes. It costs the propagated up splitting; i.e., every ancestor node has to be split. The number of nodes, thus, affected is the same as the height of the G⁺-tree, $1 + \log_m \left\lfloor \frac{N}{m_b} \right\rfloor$. Therefore, the total number of I/O for an insertion, in the worst case, is the times of I/O for finding the exact bucket node, the times of I/O for the propagating split, and the times of creating new nodes for each propagating split. Thus, three times of the height of the G⁺-tree, $3\times(1 + \log_m \left\lfloor \frac{N}{m_b} \right\rfloor)$.

Like the insertion, deletion has to go from the root of the G⁺-tree down to the bucket node and spend the $1 + \log_m \left\lfloor \frac{N}{m_b} \right\rfloor$ times of I/O before deleting a point datum. The worst case of this deletion occurs when the bucket node has exact $m_b$ (minimum capacity) data and all of its ancestor nodes and ancestors' siblings have exact m (minimum capacity) data. That means that it needs propagated deletions to the root. The re-distribution of the data affects 3 nodes (2 siblings and itself); i.e., the number of I/O is three times of the G⁺-tree's height. Therefore, including the initial nubmer of I/O for searching, totally the number of I/O is four times of the tree height; i.e., $4\times(1+\log_m \left\lfloor \frac{N}{m_b} \right\rfloor)$.

In respect of range query, its number of I/O depends on the size of the query. However, since the G⁺-tree keeps partial spatial locality, the overhead of getting data will not be too high. The worst case for a range query occurs when the range covers the cross of the nodes (Fig. 10). In this circumstance, the query range overlaps $2^d$ non-continuous parts (d is the dimensionality). In other words, the overhead (excluding the bucket nodes

accesses) of each part goes from the root to the first bucket node of that part. Thus, the total overhead for the range query is $2^d \times (\log_m \lfloor \frac{N}{m_b} \rfloor)$ in the worst case.

In terms of G-tree, it has the same structure of internal node with G⁺-tree. Hence, the distinction between these two data structure is on the bucket nodes. If the data are uniformly distributed in the data space, both of the G-tree and G⁺-tree have nearly the same space utilization in the bucket node. Meanwhile, they can have similar number of bucket nodes and internal nodes; therefore, they both can have similar time complexity for the data retrieving operations.

However, if the data are not well distributed in the data space, the G-tree might suffer the low space utilization and lose well performance. In G-tree, the worst case of space utilization occurs when the data is skewly distributed to be split for each full node coming with n nodes of minimum capacity 1. Then, the space utilization of a bucket node is $(1 \times n + M_b)/ (n \times M_b)$. The utilization will become very low when the n is getting bigger. For example, when n is 10, $M_b$ is 25, the space utilization is $(10+25)/(10 \times 25) = 35/250$ which is 14%. The low space utilization of G-tree might make its tree height higher and waste the disk space. Therefore, in the worst case, the number of G-tree's I/O might be more on search, insertion, and deletion than G⁺-tree's.

In addition to the low space utilization, the G-tree's range query has very low performance. G-tree spend only once on searching down from the root to the bucket (the first bucket node inside the query range). Then, G-tree traverses the neighborhood till it reaches the upper bound of the range query. In some worse cases which the data are cumulated in the left upper and lower right parts of the data space. From the partition number scheming, the order of the right, upper is higher than that of the left and lower

part in the data space. A range query is overlapping on the cross of the partitions, it goes through almost all the grids in the left upper and the right lower parts before reaches the right upper partition. Figure 10, a range query almost traverses all the partitions, i.e., visits nearly all the bucket nodes in the G-tree.

In respect of the space complexity, we can find $G^+$-tree is better than G-tree. Meanwhile the space complexity affects the time complexity of G-tree. Therefore, in view of the time complexity analysis, we can clearly comment that $G^+$-tree is superior to G-tree.

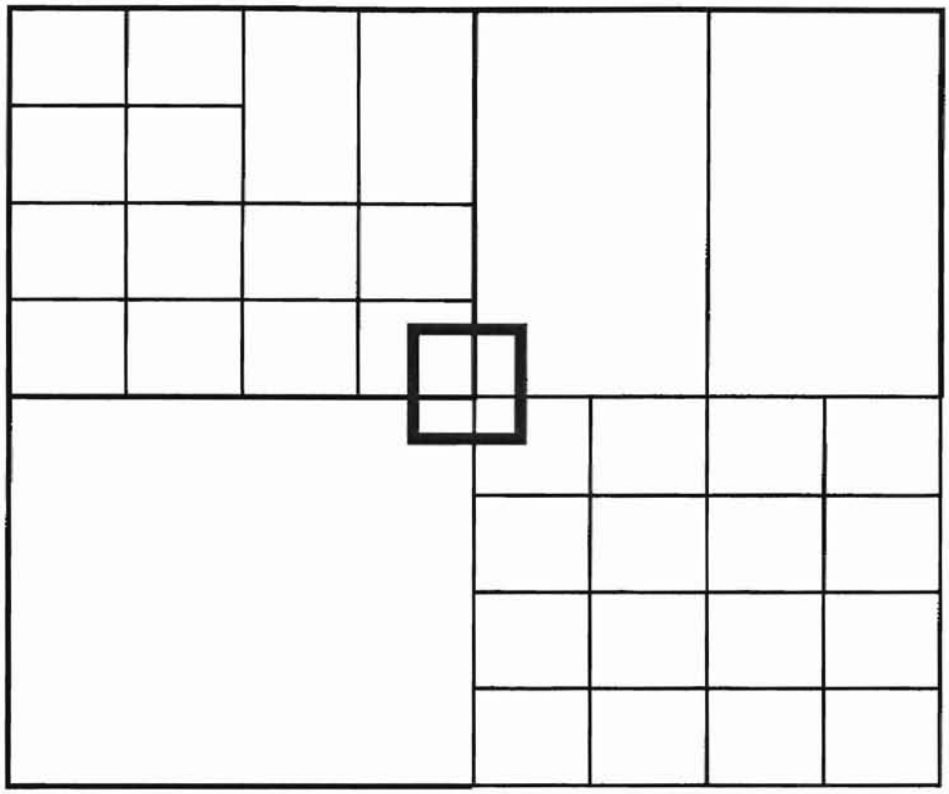

Fig. 10 An example of range query

If the query range is at the center and cross several parts

(a) For $G^+$-tree, the overhead is 4 times of the tree height because the range belongs to 4 partitions which are close in the space data but far away from the tree's bucket nodes..

(b) For the G-tree, this is an example of the worse cases. It goes all of the left upper and right lower grids for the query as it is optimistically believed to keep the property of spatial locality.

# CHAPTER V

# CONCLUSION

A new spatial data management structure named G$^+$-tree is proposed. It has successfully improved the performance of the G-tree data structure when data are skewly distributed in the data space. Even in the situation of uniform data distribution, G$^+$-tree shows not much less efficient than G-tree. One important feature is that G$^+$-tree eliminates the sparse nodes in G-tree. Hence, no any node in G$^+$-tree is sparse. This makes each node more compact and reduces the total number of the nodes in the G$^+$-tree. Furthermore, the nodes access (I/O) can be greatly reduced, especially in the situation of a very skew data distribution.

In addition, G$^+$-tree has improved the range query of the G-tree. The latter could be very time-consuming to retrieve range data because it thinks all the data within the range are stored in the close nodes but this is not true. The G-tree structure can keep the property of locality of the data partially, not completely. In the worst condition, the G-tree may need to traverse nearly all the nodes to get a very small range of data. G$^+$-tree, instead, takes the advantage of the partial locality kept in its nodes and also considers the situation that the close data may be stored in the far away data nodes.

In short, G$^+$-tree eliminates the possible degeneration of G-tree with a little bit price. Thus, it is much better than G-tree as a multi-dimensional indexed tree for large

databases, especially, when data are not well distributed.

# GLOSSARY

This glossary includes terms and definitions that are mentioned in this thesis.

**B-tree of order m**. A multi-way search tree with these properties: (quoted from [FOL92])

1. Every node has a maximum of m descendents.
2. Every node except the root and the leave has at least $\lceil m/2 \rceil$ descendents.
3. The root has at least two descendents (unless it is a leaf).
4. All of the leaves appear on the same level.
5. A non-leaf page with k descendents contains k-1 keys.
6. A leaf page contains at least $\lceil m/2 \rceil$-1 keys and no more than m-1 keys.

The power of B-tree lies in the facts that: they are balanced (no overly long branches); they are shallow (requiring few seeks); they accommodate random deletions and insertions at a relatively low cost while remaining in balance; and they guarantee at least 50 percent storage utilization.

**BANG file**. Balanced and Nested Grid file. It is an interpolation grid file. It partitions the data space into block regions by successive binary division. Then, organize the block regions into a tree like B-tree.

**GBD tree**. General BD tree. A data structure for spatial database. It uses a binary numerical scheme when split a partition. Then, organize the partition number into a B-tree for indexing.

**Grid file**. A data structure that partitions the data space into grid structures with a directory to maintain those grids for data access.

**G-tree**. Grid tree. A data structure which combines Grid file and B-tree in a special manner to index point data.

**$G^{\top}$ tree**. Evenly distributed Grid tree. This is a new data structure derived from G-tree(Grid tree) for good storage utilization and retrieval efficiency.

**Hashing**. A technique for generating a unique address for a given key. It is used to rapidly access data record.

**K-D Tree**. K denotes the dimensionality of the space being represented. It is a binary search tree with the distinction that at each tree level a different attribute value is tested to determine the direction in which a branch is to be made.

**K-D-B tree**. A data structure adopts B-tree and each node being assigned an adaptive k-d tree partition.

**R-tree**. A data structure for multi-dimensional index. It uses minimum bounded rectangle to denote an object in the spatial space, then, adopts B-tree to maintain those rectangles.

**$R^*$-tree, $R^+$-tree**. The variants of R-tree. They use different split methods for overflowed nodes to obtain better performance.

**X-tree**. A data structure for improving the $R^*$-tree. It is suitable for high dimensional data space.

## REFERENCES


[BAN95] Bang, K. S. and Lu, H. "SMR-Tree: An Efficient Index Structure for Spatial Databases," Proceedings of the 1995 ACM Symposium on Applied Computing," Nashville, Tennessee, February, 1995, 46-50.

[BEC90] Beckmann, N. and Kriegel, H. P. "The R*-tree: An Efficient and Robust Access Method for points and Rectangles," Proceedings of the SIGMOD Conf., Atlantic City, June, 1990, 322-331.

[BEN75] Bentley, J. L., "Multi-dimensional binary search trees used for associative searching." Commun. ACM. vol. 18, 9(Sep. 1975), 509-517

[BER96] Berchtold, S., Keim, D. A., and Kriegel, H.P. "The X-tree: An Index Structure for High-Dimensional Data" Proceedings of the 22nd VLDB Conf., Mumbai (Bombay), India, September, 1996, 28-39.

[BRE93] Breene, L. A. "Quadtrees and Hypercubes: Grid Embedding Strategies Based on Spatial Data Structure Addressing," The Computer Journal, v. 36, no. 6, 1993, 562-569.

[CHE92] Cheng, X., Lu, H. and Hedrick, G. E. "Searching Spatial Objects with Index by Dimensional Projections," Proceedings of the ACM 1992 Symposium on Applied Computing, 1992, 217-223.

[FOL92] Folk, M. J., Zoellick, B. File Structures, A Conceptual Toolkit, Addison Weselsy, 2nd ed., 1992,

[FRE87] Freeston, M. "The BANG File: A New Kind of Grid File", SIGMOD RECORD, 16, 3(1987), 260-269.

[GAE98] Gaede, V. and Günther, O. "Multidimensional Access Method" ACM Computing Surveys, v. 30, no. 2, 1998, 170-321.

[GUN94] Günther, O. and Lamberts, J. "Object-oriented Techniques for the Management of Geographic and Environmental Data", The Computer Journal, v. 37, no. 1, 1994, 16-25.

[GUN97] Günther, O. and Gaede, V. "Oversize Shelves: A Storage Management Technique for Large Spatial Data Objects", International Journal of Geographic Information Systems, v. 11, no. 1, June, 1997, 5-32.

[GUT84] Guttman, A. "R-Trees: A Dynamic Index Structure for Spatial Searching", Proceedings ACM SIGMOD, June, 1984, 47-57.

[HOE92] Hoel, E. G. and Samet, H. "A Qualitative Comparison Study of Data Structures for Large Line Segment Databases", Porc. ACM SIGMOD conf., San Diego, CA(1992) 332-342.

[HOS92] Hosur, N., Lu, H. and Hedrick, G. E. "Dynamic Addition and Removal of Attributes in BANG files," Proceedings of the ACM 1992 Symposium on Applied Computing, 1992, 217-223.

[KUM94] Kumar, A. "G-Tree: A new Data Structure for Organizing Multidimensional Data" IEEE Trans. Knowledge and Data Eng., 6, 2(1994), 341-347.

[LOM92] Lomet, D. "A Review of Recent Work on Multi-attribute Access Methods", SIGMOD RECORD, 21, 3(1992), 56-63.

[NIE84] Nievergelt, J., Hinterberger, H., and Sevcik, K. C. "The Grid File: An Adaptable, Symmetric Multikey File Structure", ACM Trans. Database Syst., 9, 1(1994), 38-71.

[OHS90] Ohsawa, Y. and Sakauchi, M. "A New Tree Type Data Structure with Homogeneous nodes Suitable for a Very Large Spatial Database", Sixth International Conference on Data Engineering, 296-303.

[ROB81] Robinson, J. T. , "The KDB tree: A search structure for large multi-dimensional dynamic indexes," in Proc. ACM SIGMOD Conf., Ann Arbor, MI, Apr. 1981, 10-18.

[ROT74] Rothnie, J. B. and Lozano, T., "Attribute based file organization in a paged memory environment," Commun. ACM, vol. 17, no. 2(Feb. 1974).

[SAM90] Samet, H. The Design and Analysis of Spatial Data Structures, Addison Wesley, 1st ed., 1990.

[YEH90] Yeh, S. S. "BANG File Concurrency," Oklahoma State University Master of Science Thesis, 1990.

VITA

Hung-Chi Su

Candidate for the Degree of

Master of Science

Thesis: $G^+$-TREE: A SPATIAL INDEX STRUCTURE

Major Field: Computer Science

Biographical:

Personal Data: Born in Taiwan, Republic of China, January 25, 1964, the son of Yueh-Chau Su and Li Su-Chei.

Education: Received Bachelor of Science Degree in Chemical Engineering from National Chen-Kung University, Taiwan, R.O.C. June, 1986;
Completed requirements for the Master of Science degree at Oklahoma State University in July, 1999.

Professional Experience: Programmer, Shin-Kung Computer Service, Inc., Taiwan, R.O.C., from February, 1989, to May, 1990; System Analyst, Hess chain-book-store, Taiwan, R.O.C., from June, 1989, to October, 1990; Best Color Enterprise Co., Taiwan, R.O.C., from November, 1990 to July, 1993.