IMPLEMENTATION AND EVALUATION OF BALANCED AND NESTED GRID (BANG) FILE STRUCTURES

BY

TIONG-HU LIAN

Bachelor of Science in Engineering

National Taiwan University

Taiwan, R. O. C.

1983

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
December, 1988

Thesis
1988
L693i
cop. 2

Oklahoma State Univ. Library

IMPLEMENTATION AND EVALUATION OF

BALANCED AND NESTED GRID

(BANG) FILE STRUCTURES

Thesis Approved:

Donald D Fisher

Thesis Adviser

Huizhu Lu

[illegible]

Norman N. Durham

Dean of the Graduate College

1318971

ACKNOWLEDGMENTS

I would like to express sincere gratitude to my thesis adviser, Dr. D. D. Fisher, for his suggestion, assistance, patience, and encouragement through this study. I also wish to express my deep appreciation to my committee members, Dr. H. Lu, and Dr. K. M. George for their contributions and advice, and Dr. G. E. Hedrick for substituting during my oral examination.

I wish to express my gratitute to the Department of Computing and Information Sciences of Oklahoma State University and Dr. H. Lu for providing me a teaching assistantship.

A special thank goes to my parents, Mr. and Mrs. Tjin-Siong Lian, and my brothers and sisters for their help in every aspect.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

In database systems it is often reguired to make a retrieval based on a set of keys, which is called multikey retrieval or associative retrieval. The method of retrieving the records by this type of query is a complex operation since it requires the retrieval of a set of records that satisfies given attributes. In this thesis we are concerned with a large database where all records have to be distributed among disk blocks in the secondary storage. A query to a set of records that matches the query is completed by retrieving all disk blocks where the records are located. Therefore, the efficiency of the query depends on the database organization within the secondary storage.

Currently there are several file structures that are proposed to deal with multikey retrieval such as inverted files or secondary index files for those fields or attributes which may be used as access keys during the query, multidimensional search trees or k-d trees[1,2] and various multikey hashing file structures. However, these file structures have some shortcomings. A retrieval of the records that match the query in the inverted file structure requires an excessive number of disk accesses. If the distribution of records is not uniform, then the k-d tree is highly unbalanced. In order to keep it balanced, a costly restructuring of the tree may be required. The hashing file structure handles the collisions by using an overflow bucket. This design method increases the number of disk accesses during a query, especially a query to a record which does not exist.

In this thesis we compare two types of index-based file structures, the grid file[14] and the BANG file[7] structure, used to organize large, dynamic, k-dimensional records efficiently. By large it is meant that all the records and most of the index must be stored in the secondary storage. By dynamic it is meant that insertions and deletions of records are intermixed with the queries.

The grid file is a file structure that organizes the data space of a given set of records. Each indexed attribute is handled symmetrically. Some properties are expected from the grid file structure.

1. High adaptability to the change of data distribution. A disk block can be split or merged in response to the insertion or deletion of a record.
2. Efficient query. An exact match query can always be completed in two disk accesses; furthermore, an efficient region query is obtained because the grid file structure preserves the neighborhood property of the records.

3. Reasonable storage utilization.

The BANG file is a file structure that attemps to eliminate the disadvantages of the grid file structure. We expect the following additional advantages of the BANG file.

1. The size of the directory is more compact. In the grid file structure, the size of directory grows rapidly because multiple directory elements point to the same data bucket. In order to eliminate the overhead of these pointers, the BANG file structure allows a region to be nested inside another region where the shape of the region is not necessarily convex.
2. The BANG file structure maintains a higher storage utilization. The distribution of records within the data buckets is more uniform because the partition algorithm tries to find a boundary that partitions the overflowing data bucket into two almost balanced data buckets. This property provides two advantages:
   a. a range query can be completed with less disk accesses;
   b. less splitting and merging.
3. The merging algorithm is simpler since no deadlock detection is required.

In the next section several structures that handle multidimensional records are reviewed. The concept and design strategy of the grid file and BANG file are discussed in Chapters 3 and 4. In Chapter 5 we present the comparison between 3 dimensional grid file and BANG file in space utilization and range query. In order to show the advantage of the BANG file over the grid file, we generate a set of 3 dimensional records where the values of each attribute follow a Poisson distribution. In this chapter we discuss the work needed to modify the 3 dimensional implementation into an n dimensional implementation where n > 3. The final conclusion of this thesis and suggestions for further study are given in Chapter 6.

CHAPTER II

LITERATURE REVIEW

File structures that provide multikey access to records are of interest in various fields of data processing. For example, in an enrollment file which contains information about the students of a university, it may be desirable to search for all international students who are majoring in engineering and have 3 <= GPA <= 4. The design of a file structure that allows efficient access to the records in multidimensional data (each record is identified by several attributes) is significantly more difficult than in one dimensional data, since natural orders of multidimensional data do not exist.

In practical applications the most common method for multikey access to data is to construct an inverted file, i.e., an independent index or directory file for each attribute. However, for associative queries involving more than one key, e.g. region query, it may become necessary to access a number of inverted files and, further, to perform costly intersections in order to obtain the records that satisfy the query. Another disadvantage of this method is the cost to modify the index files during an insertion or deletion.

Several multikey file structures have been proposed that avoid the deficiencies of the inverted file by combining all keys into a single access path, so that the same structure handles all keys. In general multikey file structures can be classified according to whether they are static or dynamic and whether they are based on comparative search or address computation. In the following we discuss some examples of multikey file organizations based on comparative search, such as quadtrees[6], k-d trees [1,2], multidimensional B-trees, and k-d-B trees, and some multikey file orqanizations based on address computation structures such as interpolation based index maintenance, and multidimensional linear dynamic hashing.

Quadtrees[6] are a generalization of binary trees for the treatment of data with inherently two-dimensional structure. Each node of the quadtree stores one record and has up to four sons. The root of the tree divides the universe into four quadrants, namely NE, NW, SW, and SE. Insertion of a new record into a quadtree is based on the same philosophy that governs insertion into binary trees. At each node, a comparison is made and the correct subtree is chosen for the next test. The position to insert the new record is found after the comparison with the leafnode. There are several disadvantages in the use of quadtrees to organize a dynamic multidimensional data.

1. In a nonuniform data distribution, a costly restructuring is needed to keep the quadtrees balanced.

2. Deletion of a record from quadtrees is complex. The difficulty lies in deciding what to do with the subtrees that were attached to the deleted node.
3. The number of sons of a node for k dimensional data is 2^k. If k is large, the amount of memory required to store the nodes and the number of pointers is very large; furthermore, many null pointers for the leafnodes. In the balance quadtrees with level = x, the number of null pointers is as much as k^{x-1}.

A k-d tree[1,2] is a natural generalization of the well-known binary search tree to handle multikey records. In a standard binary search tree, a single key is used to decide whether a record lies to the left or the right of a node. In a multidimensional search tree(k-d tree) this decision is made based on different keys. Consider that each record in the file has k keys, K_1, K_2, .., K_k. On the first level of the tree the decision to go right or left when inserting a new record is done by comparing the first key of the new record with the first key of the record stored at the root of the k-d tree. Consider the root of the tree has level l = 0, then the key used to make the decision is (l + 1) mod k. Therefore, for any node x with the key discriminator K_j, all nodes in the left subtree of x have K_j values less than x's K_j value and likewise all nodes in the right subtree have greater K_j value. There are some shortcomings of this file structure in dealing with a large and dynamic database.

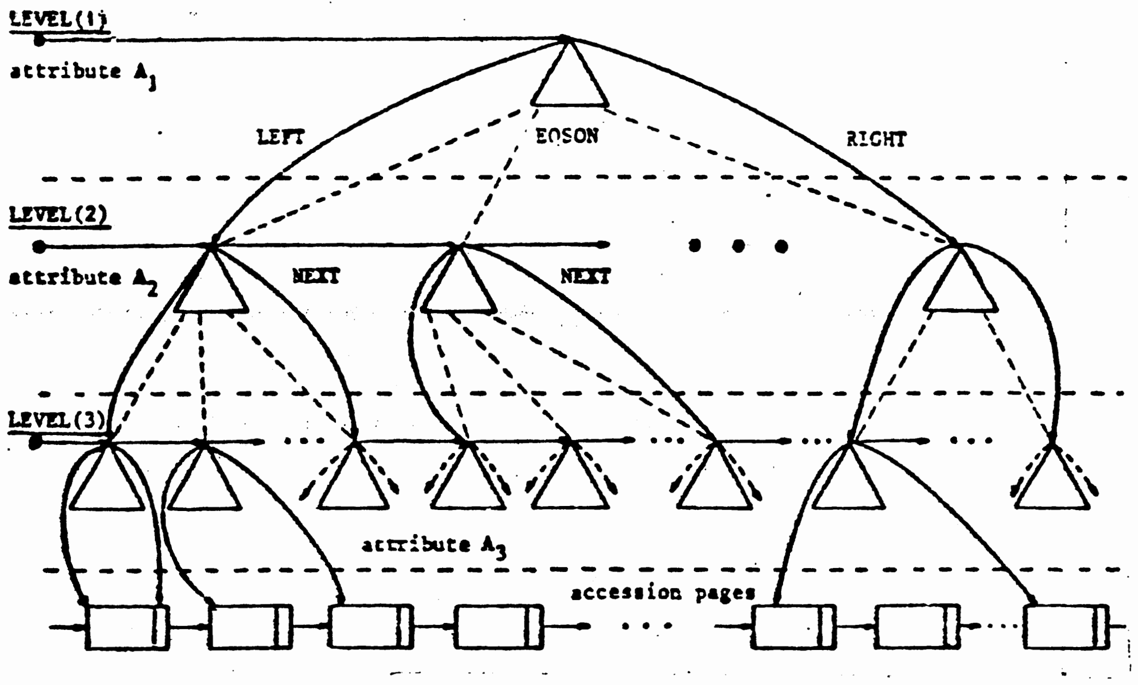


Figure 1. 3 Dimensional MDBT.[11]

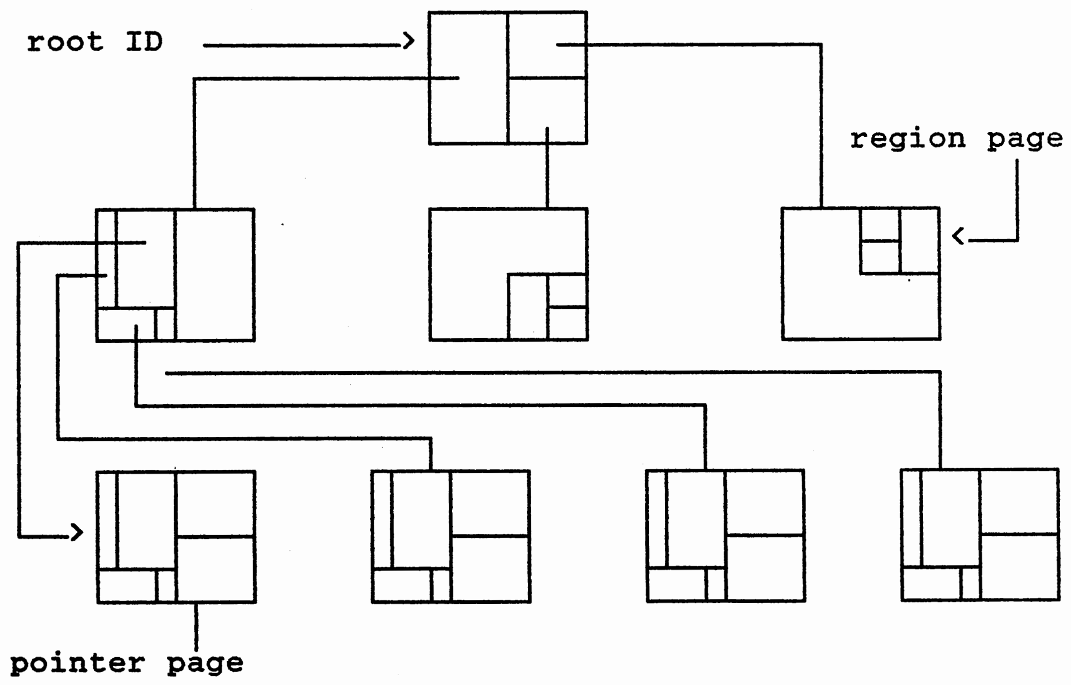


Figure 2. Example of 2-D-B-tree[22].

1. In nonuniform data distribution, the tree may become unbalanced. A costly restructuring algorithm is needed to keep the tree to be balanced.
2. In a large database, the size of an index is large and should be stored in secondary storage. A search for a record or set of records needs several disk accesses.

A multidimensional B-tree[18,24] is a file structure that organizes multidimensional data by using the B-tree where each node of the B-tree itself is a B-tree. Figure 1 shows an example of 3 dimensional data organized by multidimensional B-trees. Each of the n attributes of the data to be indexed is represented by a separate level in the tree directory. The values of the i-th attribute are organized as B-trees of order m_i, where m_i may vary according to the length of the values of the i-th attribute. Each attribute value x_i in level i has a pointer to a B-tree at level i+1 which stores all different values of attribute A_{i+1} with common value x_i for attribute A_i. This set of values of attribute A_{i+1} with common value x_i is called a filial set. In order to allow direct access to any level i of a multidimensional B-tree organization, all filial sets at level i are linked together and a pointer LEVEL[i] points to the root of the first filial set of each such linked list. The root node of each B-tree contains three additional pointers, called NEXT, LEFT, and RIGHT pointer. The NEXT pointer points to the root of the next B-tree in the list to provide a sequential access along the list of B-trees in a level. The LEFT and RIGHT pointers of a B-tree X point to two filial sets in the linked list. All parents of the filial sets between these two filial sets reside in X. In a B-tree, it is assumed that each node of the tree is assigned to one disk block. If this restriction is enforced, the index storage utilization may be very low and thus the number of index pages very high. In order to maintain reasonable storage utilization, several nodes may be assigned into the same page. One assumption for the multidimensional B-tree is that the distribution of data is uniform. This assumption is to ensure that all B-trees on the same level have the same height which quarantees a minimal retrieval time of $O(\log_m N)$[18] for an exact match query of the multidimensional B-tree. In the case of a nonuniform distributed database, the retrieval time may be as much as $O(k * \log_m N)$.

Another file structure that combines the properties of B-trees and K-D-trees is the K-D-B tree[22]. K-D-B-trees, like B-trees, are multiway trees with fixed-size nodes that are always totally balanced in the sense that the number of nodes accessed on a path from the root node to a leaf node is the same for all nodes. It is expected that the multidimensional search efficiency of balanced K-D-trees and the I/O efficiency of B-trees should both be approximated in K-D-B-tree. K-D-B-trees consist of a collection of pages and a variable root ID that gives the page ID of the root page.

There are two types of pages in a K-D-B-tree.

1. The pointer pages which store the leaf nodes of the tree. These pages contain pointers to those records which correspond to a region in k-dimensional space.
2. The region pages that reflect the partitioning of a region into nonoverlapping subregions.

The root of the tree represents the entire data space.

Like K-D-trees, K-D-B-trees partition the search space into two subspaces based on comparison with some element of a single domain. There are several methods that can be used to select the dimension for splitting. One way is to choose the dimension cyclically. This cyclic method might be modified if something is known about queries. In the case that most of the partial range queries specify a certain dimension only, then it would be desirable to split that dimension several times before splitting the other dimensions. Figure 2 depicts an example of 2-D-B-tree.

In his report, Kriegel[11] shows the performance comparison between 4 file structures in dealing with multidimensional databases. They are inverted file, multidimensional B-tree, K-D-B-tree, and grid file structure. The result of his experiments shows that the inverted file has the worst performance both in space requirement and response time for the queries and the multidimensional B-tree ranks third. The performances of the K-D-B-tree and the grid file depend on the data distribution scheme. For a database where the attributes are nonuniformly distributed, the grid file has

TABLE I

PERFORMANCE COMPARISON

	rank / struc.	a	Uniform b	Uniform c	Non-uniform b	Non-uniform c
1	Inv. files	3	4	4	4	4
2	MDBT	2	3	3	3	3
3	K-D-B-trees	2	2	2	2	1
4	Grid file	1	1	1	1	2

a represents space utilization.
b represents insertion, deletions, and exact match queries.
c represents partial match queries and range queries.
1 represents the best performance and 4 represents the worst performance.

best performance relative to space requirements, and processing times for insertions, deletions and exact match queries. The K-D-B-tree performs best for partial match queries and region queries. For a database with uniformly distributed and independent attributes, the overall performance of the grid file overcomes the performance of the K-D-B-tree. However this type of database is unlikely in real life. Table 1 shows the summary of this result. In this thesis we also introduce the BANG file, a modification of the grid file structure. Based on its design strategy, we expect that this file structure is more adaptable to the nonuniform distributed database.

Multidimensional linear dynamic hashing[19] and interpolation-based index maintanance[3] are multikey file structures which are derived from linear hashing[12]. In the classical hashing method, a collision is handled by overflow buckets. However, if all collisions are resolved by overflow buckets, the access performance of the hashing method deteriorates rapidly. To avoid this disadvantage, linear hashing resolves the problem of collisions by dynamically modifying the current hashing function $h_i(x)$ to $h_{i+1}(x)$. Suppose the insertion of record r into bucket $h_0(r)$ leads to a collision. Instead of storing r in the overflow bucket, the linear hashing method splits the records in bucket $h_0(c)$ into two subsets by utilizing the split function h_1.

Let D be the key space. $h_0 : D \dashrightarrow \{0, 1, \ldots, n-1\}$ be the function that is used to load the file. The functions $h_1, h_2, \ldots, h_i, \ldots$ are called split functions for h_0 if they obey the following requirements:

[range condition]: $h_i : D \dashrightarrow \{0, 1, \ldots, 2^i n - 1\}$

[split condition]: for each r in D

$$h_i(r) - \begin{cases} h_{i-1}(r) \text{ or} \\ h_{i-1}(r) + 2^{i-1} * n \end{cases}$$

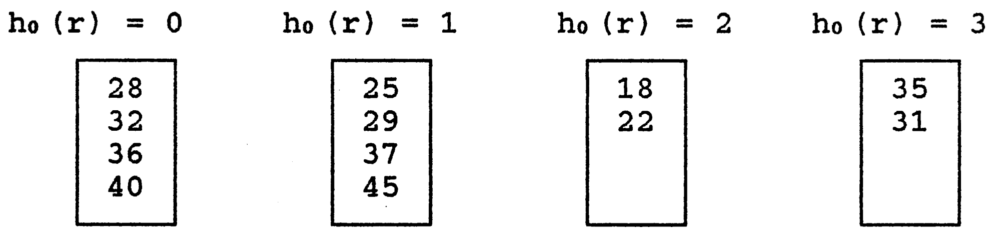
$i = 0, 1, 2, \ldots$

Two variables are maintained for this process:

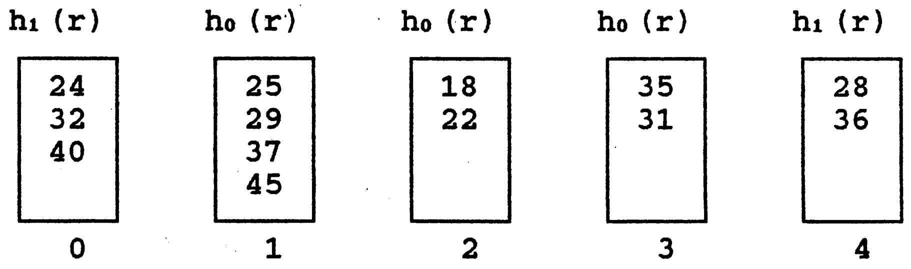
NEXT	points to the next chain to be split
LEVEL	represents the number of times the address space has doubled in size.

The initial value of both variables are set to 0 and update as follows

```
NEXT = (NEXT + 1) mod (n * 2^l)
if (NEXT == 0) then L = L + 1
```



(a) Before Splitting.



(b) After Splitting of bucket 0.

Figure 3. Linear Hashing.

For example, a file with records 25, 35, 18, 22, 28, 29, 32, 36, 40, 31, 37, 45 is organized by using the linear hashing method with hash function $h_0(r) = r \bmod n$, where $n = 4$. The bucket capacity is $b = 4$ records. The resulting file is shown in Figure 3.a. If the record 24 is inserted, then a collision occurs. The address m where the record $r = 24$ is stored is determined as follows:

```
$m = h_0(r)$
if $m <$ NEXT then $m = h_{l+1}(r)$
```

The result is shown in Figure 3.b. As more records are inserted into the file, we eventually arrive at the situation where each of the original chains has participated in a split operation. In this case the file is as if it were loaded using hash function $h_1(r)$ and the process can be repeated indefinitely($h_2(r)$, $h_3(r)$, ...).

Multidimensional linear dynamic hashing[19] is a multikey file structure that uses the idea of linear hashing. If an insertion of a record causes a collision, one of two actions may occur.

1. If the load factor does not exceed a predefined value, then the collision is handled by an overflow bucket.
2. If the load factor exceeds the predefined value, the bucket is split and the records are redistributed among the two newly formed buckets.

A collision during insertion is handled by applying the hashing function to the key in cyclic order, i.e., when the first N collisions occur in a file consisting of N buckets, they are solved by splitting the buckets along dimension 0, as if the records consist only the key k_0. At the completion of this sequence of splits the number of buckets has doubled in size, and now if more collisions occur they will be resolved by splitting along dimension 1 until the address space doubles again. The main disadvantage of this method is the data distribution must be uniform, otherwise the overflow chains may be long and therefore reduce the performance.

Interpolation-based index maintanance[3] is an adaptation of Litwin's linear hashing. A sequence of hash functions h_0, h_1, ... is defined to support the desired operations insert, delete, update, and query. These hash functions map records to chains. Each chain is associated with a region. In order to utilize the idea of Litwin's linear hashing in multidimensional data, the multikey records is mapped into a single key records. [3] suggests a method called shuffle order. Consider $k = (k_1, k_2, ..., k_d)$ is a point in $K = [0,1)^d$. Each component k_j of k has a binary representation

$$k_j = \sum_{i \geq 1} k_{ji} 2^{-i} \quad \text{, then define}$$

$$S(k) = \sum_{i \geq 1} \sum_{1 \leq j \leq d} k_{ji} 2^{-d(i-1)-j}$$

where S(k) is mapping from $K = [0,1)^d$ to $[0,1)$. Then the record is treated as if it has only one key S(k). Similar to the multidimensional linear dynamic hashing, this method is also sensitive to the non-uniform distributed data.

CHAPTER III

GRID FILE

Introduction

Consider a file F as a collection of records. Each record R of F is an ordered k-tuple (a_1, a_2, ..., a_k), where a_i can be used as the search key to access the records in file F. The value of each attribute a_i is chosen from a linear ordered domain Di. In addition to the key values, each record may contain some information which is not used as the access key. A file that contains this type of records is called k-dimensional file. In a large database, all records should be stored in the secondary storage as a collection of disk blocks or buckets. A block is a unit of data transferred between storage devices and the main memory for processing. In a computer system, the size of block is fixed and relatively small compared to the size of the database. A query to a record or set of records is handled by retrieving it from the secondary storage. The amount of time required to read or write a block, which is the sum of seek time, latency time and transfer time, is much longer than the processing time in the main memory. Therefore, one criteria that can be used to measure the efficiency of a file structure is the number of disk accesses required for basic operations such as insertion and deletion of a record during an exact match query.

The efficiency of a region query is influenced by the storage utilization and the storage scheme of a file system. Records that match this type of query may be stored in several data buckets. A low storage utilization of the data buckets may cause these records to occupy more data buckets which means more disk accesses are needed. Another important property that may affect the performance of the region query is the distribution scheme of the records among the data buckets. A file structure that stores the records by preserving their neighborhood property provides a better performance in region query.

A dynamic file system is a file system where deletions and insertions are intermixed with the queries. An efficient file system should have the flexibility to grow and shrink dynamically and be able to adapt itself to the change of data distribution. Reference [14] suggests 4 principles that guide the design of the grid file structure.

1. Two disk accesses needed for an exact match query.
2. An efficient processing of a range query in large linearly ordered domains.
3. Splitting and merging of grid blocks involves two disk blocks.
4. Reasonable bucket utility.

Grid File Structure

Grid file is a file structure which is designed to organize a set of data by partitioning a k dimensional data space according to an orthogonal grid. The boundaries of these grids are defined by k one dimensional arrays, called scales. Each scale represents an attribute/key of the file F and the range between the first and the last elements of a scale determines the range of the corresponding attribute. Each element of the scale represents a (k-1) dimensional hyperplane that partitions the data space into two disjoint, box-shape regions.

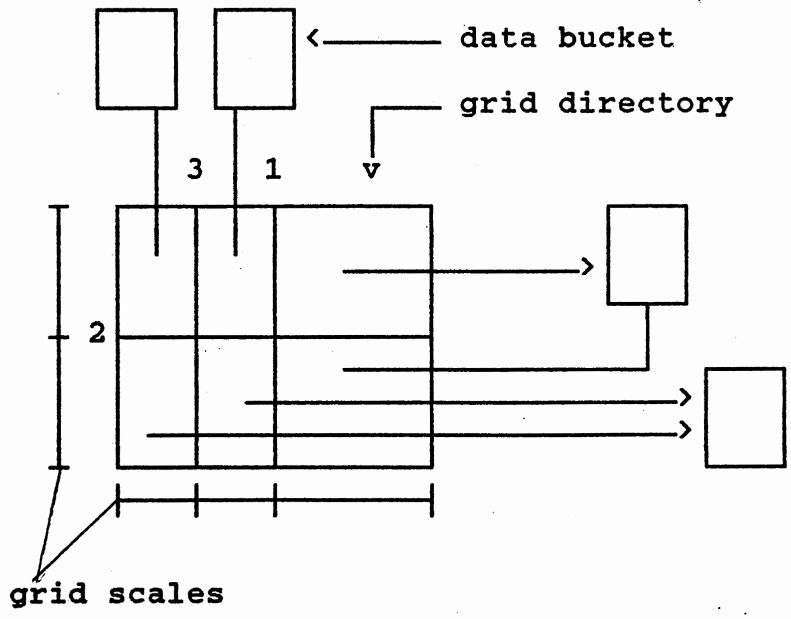


Figure 1

Figure 4. Organization of Grid File.

Figure 4 shows an example of grid file system with dimension = 2. The relation between the grid cells and the data buckets in the secondary storage is maintained in the grid directory which consists of two parts.

1. A k-dimensional dynamic array, called the grid array. Each element of the grid array represents a grid cell. The contents of a grid cell is a pointer to a data bucket which contains all records that lie in the corresponding grid cell.
2. A set of k one-dimensional arrays, called scales. These scales are used to refer to the grid array.

In a large database, where thousands of data buckets are needed to hold the entire database, the size of the grid array is likely to be very large too. To make optimal use of the main memory, the grid array must be stored on secondary storage, but the scales are small and can be kept in main memory. In order to achieve the two-disk-access property for exact match queries, all records that lie in a grid cell must be stored in the same data bucket. By enforcing this constraint, both successful and unsuccessful exact match queries can always be completed by 2 disk accesses. One for accessing the grid array and the other for accessing the data bucket.

In a dynamic database, the data and grid directory bucket may need to be modified in response to the insertions and deletions. An insertion into a data bucket may cause it to overflow. In the index-based file system, the overflow is handled by an overflow bucket. This method increases the number of disk accesses needed during an exact match query, especially for an unsuccessful query. The grid file system handles the overflow by splitting the data bucket into two. A deletion of a record may cause the content of a data bucket to drop below a certain threshold. To maintain reasonable storage utilization and to reduce the number of disk accesses required during a region query, the grid file allows a data bucket to be shared by several grid cells. A merging of two data buckets is allowed if and only if the new grid region remains convex (k-dimensional rectangular), and the contents of the new bucket after merging does not exceed a certain threshold to avoid having this bucket overflow after a few subsequent insertions.

Grid Directory

In order to make optimal use of the available main memory, the grid directory must be stored on secondary storage. All grid directory elements are distributed among several disk blocks. The method of how the grid directory elements are organized in these directory blocks affects the performance of the grid file, due to the number of disk accesses needed for neighborhood and update operations. One possible method is to store the elements of the grid directory array in secondary storage by using the conventional row/column major order. Although it optimizes storage utilization, this storage method apparently suffers from several disadvantages.

1. A costly restructuring of the entire grid directory may be needed during the splitting or merging operations.
2. The number of disk accesses during the neighborhood operations is increased, because the row/column major storage method does not preserve the neighborhood properties of the directory elements symmetrically with respect to all dimensions.

Consider the grid file system as shown in Figure 5. Assuming that the directory bucket capacity = 4, then 4 buckets are needed to store the whole directory. If a region query to the shaded area as shown in Figure 5 is issued, then all directory buckets have to be accessed in order to retrieve all records namely 2, 6, 10, 14 that match the query.

One approach[14] is to manage the grid directory with

Figure 2

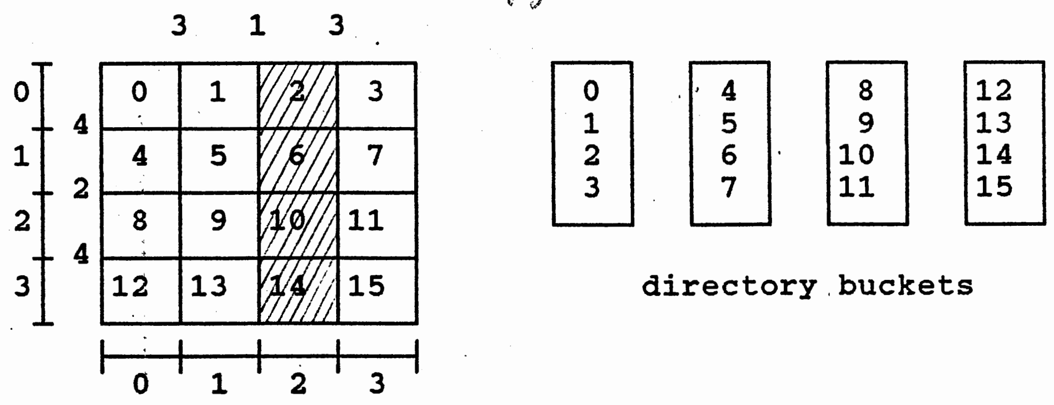
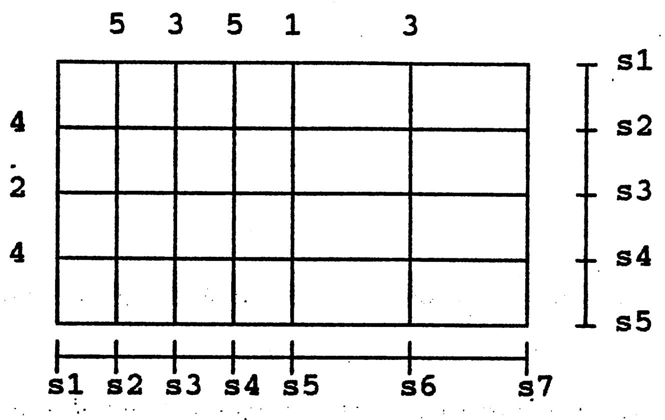
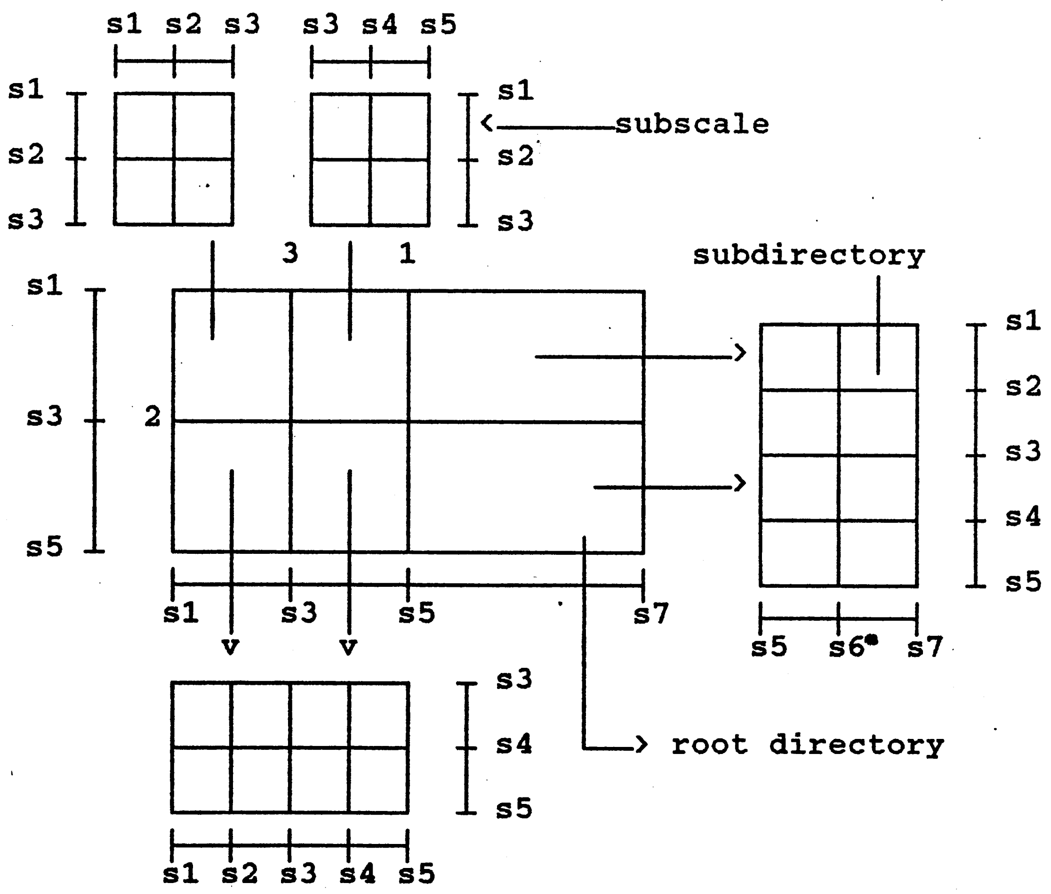


Figure 5. Single Level Directory.



a. Single Level Directory.

3.a



3.b

b. Double Level Directory.

Figure 6. Directory Structure.

another grid file. We call the directory of this grid file the root directory and the corresponding scales as root scales. The root directory is a scaled-down version of the original grid directory, in which the limit of resolution is significantly coarser. Each element of the root directory is a pointer to a directory bucket which contains the corresponding part of the original grid directory. In the following discussion, we call this directory the subdirectory and the scales corresponding to it the subscales. Figure 6.a shows the example of a single level directory and Figure 6.b shows the corresponding double level directory. The double level grid directory design strategy has some advantages with the cost of extra main memory for the root directory and root scales.

1. The split and merge operations on the grid directory are restricted to 2 disk accesses.
2. The two-disk-access principle of exact match query is preserved by storing the root directory and root scales in main memory.
3. Since the neighborhood property of the directory elements is preserved in symmetrical way, an efficient region query can be expected.

Performance

The performance of a file structure can be determined by its storage utilization and by the number of the disk accesses during a query. The grid file assures 2 disk accesses for both successful and unsuccessful exact match query by the restriction that all data points that fall into a grid region should be stored in the same disk block. In the traditional file structure which handles an overflow by the overflow bucket, an attempt to access a record may require the search along the list of overflow buckets, especially if the record does not exist. In grid file, an overflow is handled by splitting the overflow bucket and the correspondence between the data buckets and the directory elements is maintained . The same rule is also applied if the subdirectory bucket overflows.

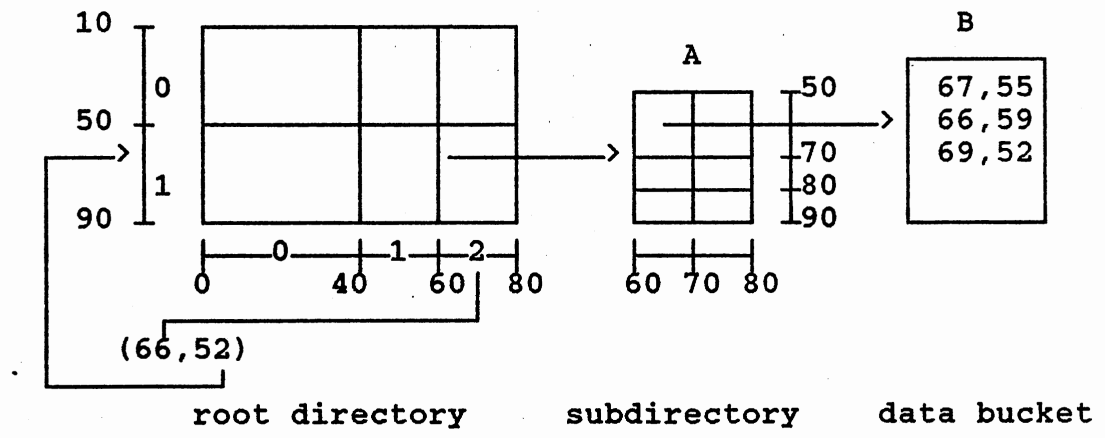


Figure 7. Accessing Method.

[illegible]

Figure 7 shows how the grid file performs the exact match query in two disk accesses. If the exact match query to find the record P with key A1 = 66 and A2 = 52 is issued, the query is performed as follows.

1. Convert the attribute value 66 into interval index 2 on the horizontal scale, and attribute 52 into interval index 1 on the vertical scale. Use these interval indices to find the correct root directory element which point to the subdirectory bucket. Since the root directory and root scale are located in main memory, no disk access is needed to perform this operation.
2. Retrieve the subdirectory bucket pointed by the root directory element which is obtained from step one.
3. Search the subscale retrieved from step 2 to find the correct subdirectory element which contains a pointer to the correct data bucket where the record P is located.
4. Retrieve the data bucket and search for record P. If it exists, then it must be located in this bucket.

This simple example shows that the exact match query can always be completed in two disk accesses.

The performance of the grid file structure is also influenced by the storage utilization of the directory and data buckets. A low storage utilization increases the number of disk blocks that have to be retrieved during a region query, since more buckets are needed to handle the records.

Update

All records in a grid file are organized by spreading them among disk blocks or data buckets according to their locations in the data space. Since the size of the disk block is fixed, an insertion into the block may cause it to overflow. In order to maintain a one to one correspondence between the grid region and the data bucket, an overflow invokes the splitting algorithm. In a double level directory grid file, the subdirectories are also stored in the disk blocks. A splitting of an overflow data bucket may cause its corresponding subdirectory bucket to be split. If a record or a directory element is deleted from the bucket, the population of the bucket may fall below a certain threshold. In order to maintain a reasonable storage utilization, a deletion, which causes the bucket to underflow, will invoke a merging algorithm. In short, a splitting of a data bucket R or a merging of two data buckets into a new data bucket R occurs at 4 levels.

1. Only the data bucket R is split or merged.
2. A region in the subdirectory B corresponding to the data bucket R has to be split or merged.
3. The directory bucket B has to be split or merged.
4. The splitting or merging of directory bucket B causes the root directory and root scale to be split or merged.

Splitting

In a double level directory grid file, an overflow of a data bucket is handled by partitioning the grid region of

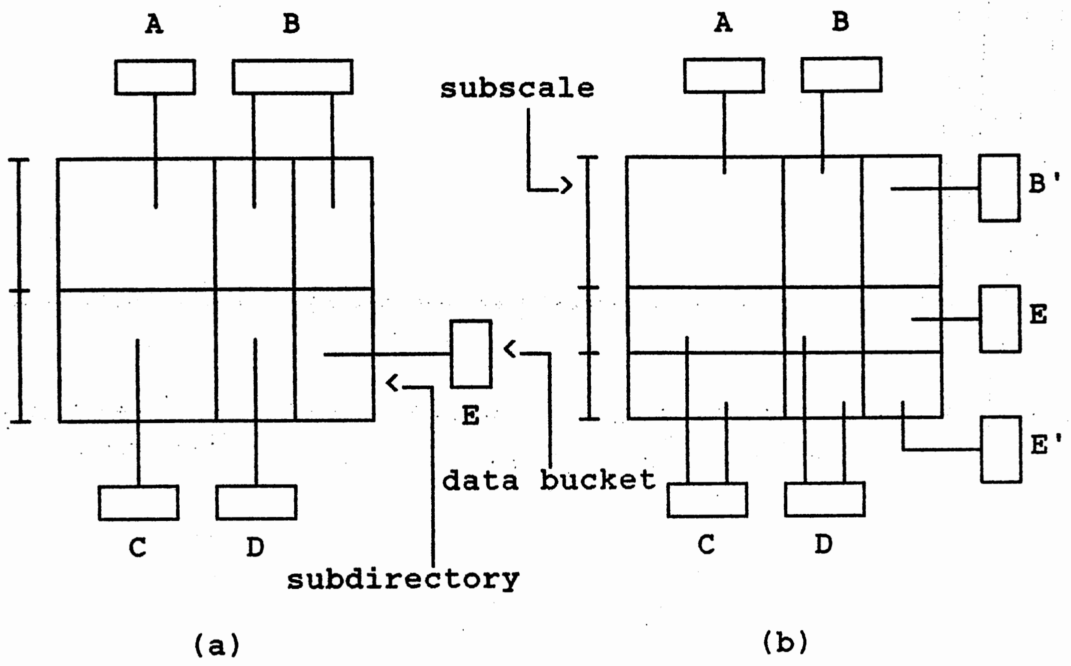


Figure 8. Splitting of Databucket and Subdirectory Region.

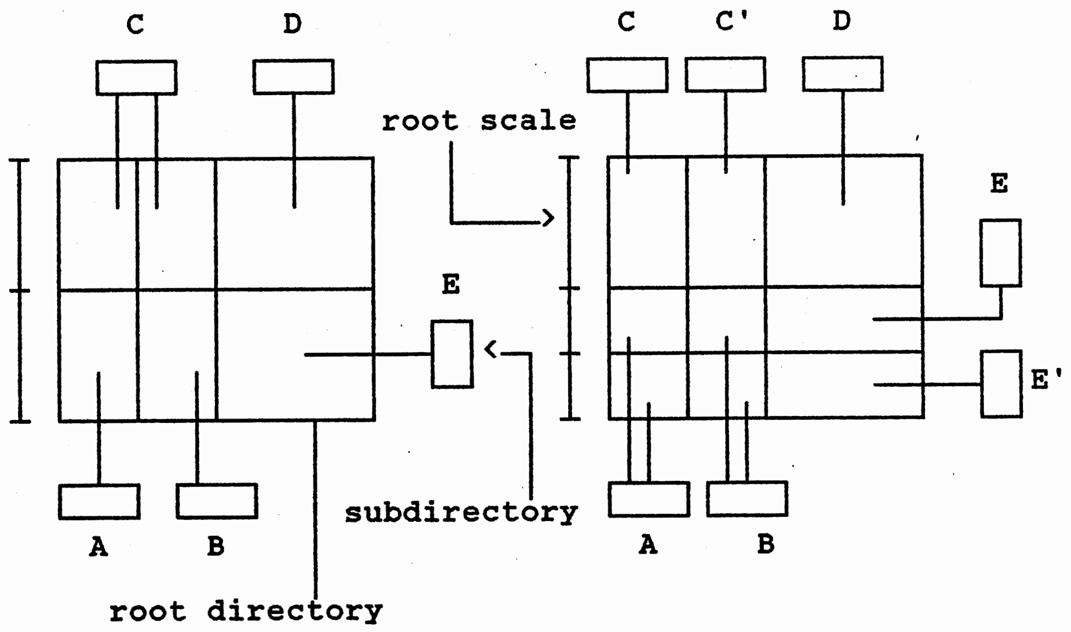


Figure 9. Splitting of Subdirectory Bucket and Root Directory Region.

the subdirectory which is corresponding to the overflowing data bucket. If this region consists of one grid cell only, a new boundary has to be inserted into one of the scales and the correspondence between the newly created grid cell and the data bucket has to be updated. For example, if the data bucket E in Figure 8 overflows, then a new boundary is inserted into one of the subscales and the data bucket is split into two new buckets E and E'. The dimension chosen for inserting a new boundary is decided in the following way.

1. Find the range r[i] of the grid region to be split for each dimension.
2. Compute the partition level l[i] for each dimension by the following algorithm.

```
  for(i=1; i<=k; i++) {    /* k-dimensional data */
    l[i] = 0;
    temp = D[i];           /*range of dimension i*/
    while(temp != r[i]) {
      temp = temp div 2;
      l[i] = l[i] + 1;
    }
  }
```

   The value of l[i] denotes the number of bisections of domain i needed to obtain an interval r[i].
3. Find the smallest partition level l[i], if it exists then i is chosen as the partition dimension.
4. If more than one dimension with minimal value of

l[i], then find a scale with the smallest number of boundary among the dimensions found in step 3.

5. Else, choose the smallest dimension number among the dimensions found in step 3 as the splitting dimension(D[1] < D[2] < ... < D[n]).

After the splitting dimension i is chosen, all records in the overflowing data bucket are redistributed. Records that fall into the left part of the new boundary are moved to the new data bucket. If the overflowing condition is removed after the splitting, i.e. at least one record is moved into the new bucket, then the new boundary is inserted into the scale i and the correspondence between the newly-formed data bucket and the grid regions is maintained. In a nonuniformly distributed database, the first partition of this chosen dimension may not remove the overflowing condition. In this case, the next dimension is chosen until the overflowing condition is removed.

If the region corresponding to the overflowing data bucket consists of more than one grid cell, some modification has to be made in choosing the partition dimension. To avoid the addition pointers overhead during the splitting, the first attempt is to choose the dimension with the existing boundary entry in its scales. For example, if the data bucket B of Figure 8.a overflows, it is split into two data bucket B and B' without introducing a new boundary into the scale.

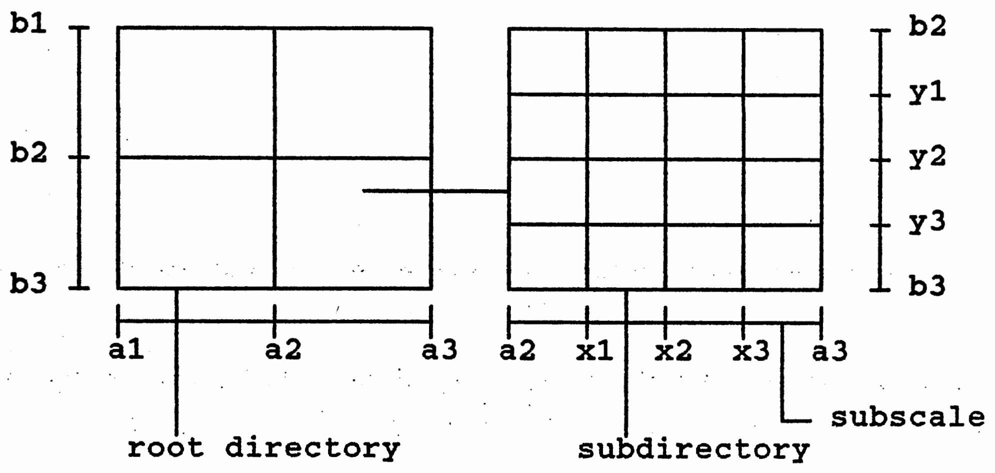
A splitting of a subdirectory region as shown in Figure

8 may cause the directory bucket itself to overflow. In this case, the directory bucket has to be split and the correspondence between the root directory and subdirectory buckets has to be maintained. Similar to the data bucket, the splitting of the directory bucket has two possible condition(see Figure 9):

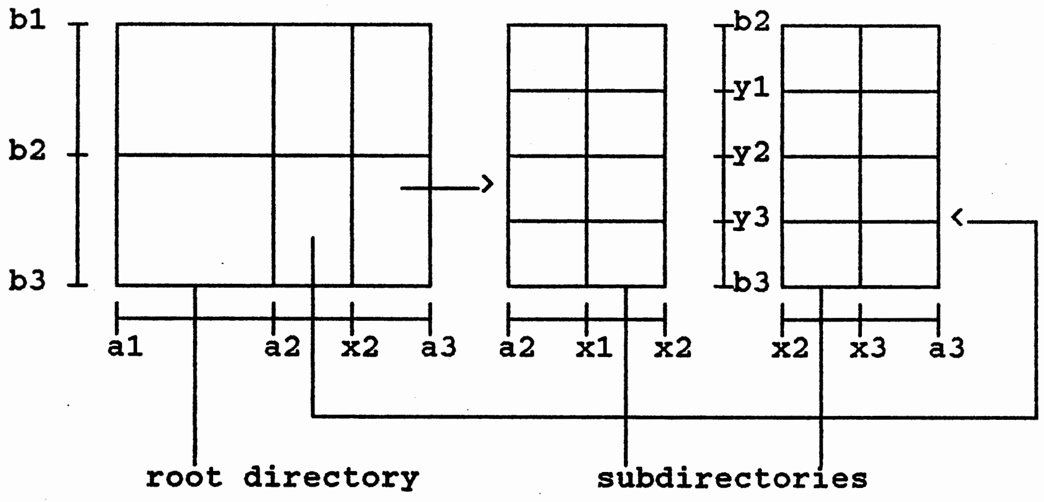
1. the overflowing directory bucket C is shared by more than one grid cell of root directory;
2. the overflowing directory bucket E is pointed by a grid cell.

The method to choose the splitting dimension discussed above can also be used for directory bucket with some additional conditions.

1. The subscales of the two new pages are created as shown in Figure 10. The old subscale of the splitting dimension is split into two new subscales. The "left" subscale is assigned to one of the new pages. All regions of the old subdirectory that lie in the "left" part of the hyperplane defined by the split boundary a2.2 are assigned into this new page. The right part of the subscale and the corresponding subdirectory regions are assigned into the other page. The subscales of other dimensions are the same with the old subdirectory.
2. A data bucket can only be shared among several grid cells of the same subdirectory. If some grid cells of the overflowing subdirectory that share the same



(a)



(b)

Figure 10. Subscales.

data bucket are redistributed among the two new pages, then the corresponding data bucket has to be split.

<u>Merging</u>

A deletion of a record from a data bucket may cause the population of the data bucket to fall below a certain threshold. To avoid the degradation of grid file performance due to the low storage utilization, a merging operation is invoked to merge an underflowing databucket with another databucket. The merging of these two buckets can be completed if and only if:

1. the resulting region corresponding to the newly formed bucket is "box-shaped";
2. no deadlock occurs after the merging;
3. the population of the data bucket after the merging does not exceed a certain threshold.

A deadlock occurs if buckets can not be merged because the resulting region would not be box-shaped. As a consequence, the average bucket occupancy may decrease and the size of the grid directory increases, which means a degradation of grid performance.

In this section we discuss two methods proposed in [14] to find the candidate with which an underflowing bucket can merge, namely,

1. the multidimensional buddy system;
2. the neighbor system.

Given an interval r1 which can be obtained by repeatedly partitioning the corresponding dimension Di. There is exactly one interval r2 such that r1 and r2 are disjoint and the union of r1 and r2 can be obtained by repeated partitioning of the dimension Di. r2 is called the buddy of r1.

A multidimensional buddy system is a method to merge a grid region with its buddy. Since there is exactly one buddy in each dimension, there are at most k candidates that can be selected as the victim for merging. In the neighbor system, a grid region can be merged with either of its two adjacent neighbors in each dimension as long as the result region is box-shaped, therefore, there are at most 2k candidates with which a data bucket can merge. An example of the buddy system and neighbor system is shown in Figure 11.

A merging of two grid regions by using the neighbor system may lead to a deadlock in a grid file of two or more dimensions(Figure 12). In a three or more dimension grid file, the buddy system merge can also lead to a deadlock (Figure 13).

The merging of two data buckets is performed by appending the records of one data bucket to the other. The correspondence between the subdirectory and the data buckets is maintained by placing the pointers of the region corresponding to the discarded bucket into the other bucket. In order to avoid overflow in the newly merged data bucket soon after a few subsequent insertions, two buckets are merged if and only if the population of the new bucket does not exceed

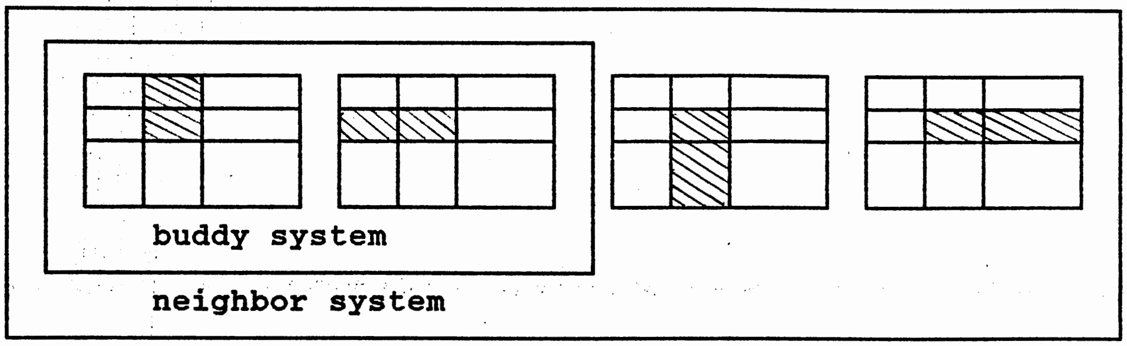


Figure 11. Buddy and Neighbor System[14].

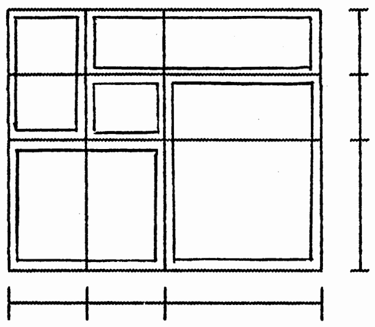


Figure 12. Deadlock in Neighbor System.

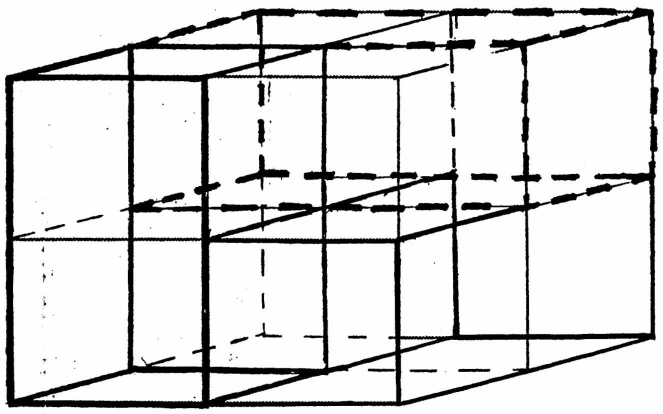


Figure 13. Deadlock in Buddy System.

a certain threshold. A lower threshold of 30 % and upper threshold of 70 % are suggested[9].

If two data buckets are merged, the boundary along which the two buckets are merged is checked. If it is no longer needed, then it is removed from subscales and the correspondence between subscales and the directory elements are updated. The removing of a boundary from the subscale may cause the storage utilization of directory bucket falls below a certain threshold. In order to maintain a reasonable storage utilization, a merging algorithm is invoked. Figure 14 shows an example of subdirectory merging.

The subscales of the subdirectory resulting from the merging are created as follows. For the merged dimension the new subscale is obtained by concatenation of two subscales of old subdirectories and all other subscales are the same

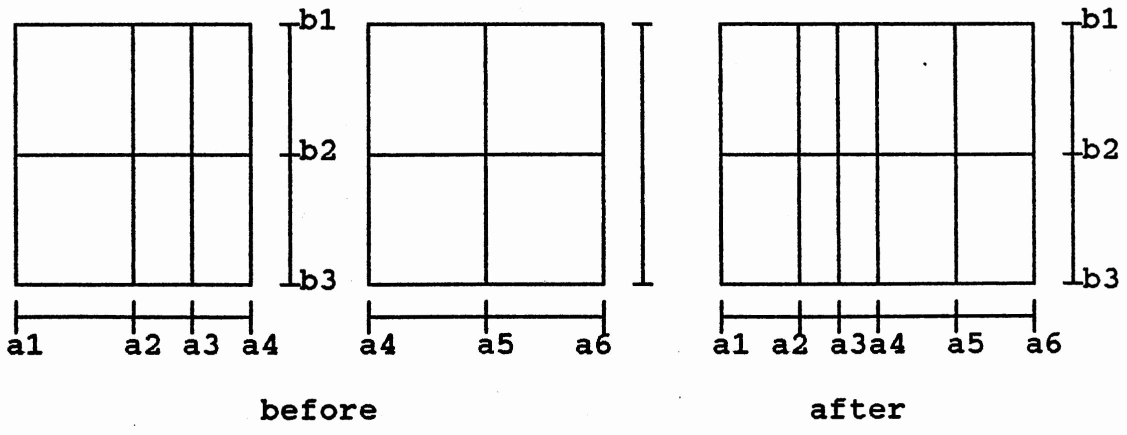


Figure 14. Directory Merging.

with the scales before merging. Figure 14 shows an example merge operation with the horizontal dimension as merging dimension.

Query

Searches are typically initiated in response to a query for the set of records. In this section we consider 3 types of query.

1. Exact match query
2. Partial match query
3. Region query

An exact match query is a query which specifies the value for each of the indexed attributes. The grid file system supports the exact match query efficiently. Each query, successful or unsuccessful, can always be completed in two disk accesses. The first disk access is to retrieve the directory bucket to find a correct directory element and the second is to retrieve the correct data bucket where the record should be located if it exists. The grid file structure does not prohibit the existence of records with the same key value for each dimension. The only restriction is that the total number of these records does not exceed the size of disk block. An example of exact match query is given in Figure 7.

A double level directory implementation of grid file preserves the neighborhood properties of the data points according to their location in data space. This property

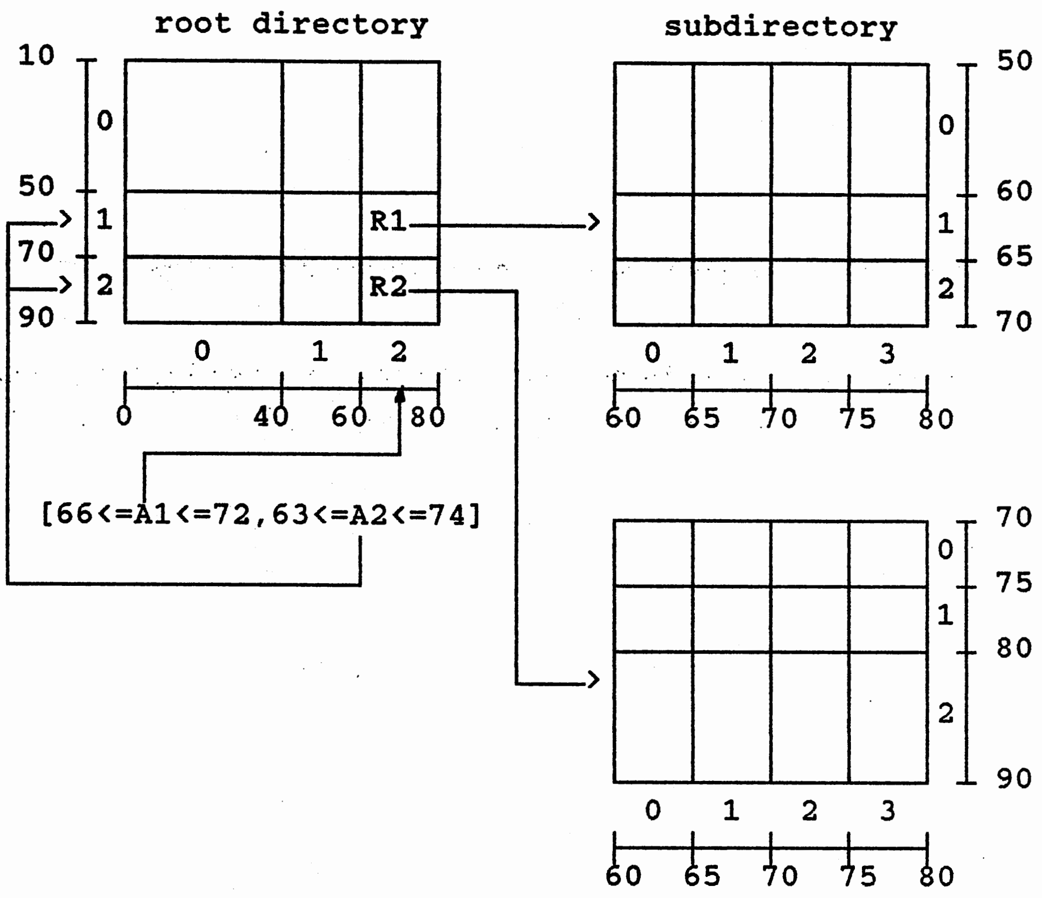


Figure 15. Region Query.

promises an efficient range query of the grid file. All data points that are neighbors in data space would be likely to be located in the same bucket. In Figure 15 we present an example of range query.

Consider a 2 dimensional record space with attribute A1 with domain 0 to 80 and attribute A2 with domain 10 to 90. Assume that the root and subdirectories are partitioned as shown in Figure 15. A range query with specification [66<=A1<=72, 63<=A2<=74] is executed as follows. The key value 66 and key value 72 of A1 are converted into interval 2 of the horizontal dimension. The key value 63 and 74 of A2 are converted into interval 1 and interval 2 of the vertical dimension. Based on these intervals, the subdirectory blocks corresponding to the regions R1 and R2 of root directory are retrieved. The same method is used to find the regions where the records that match the query lie in the subdirectory. All data buckets that correspond to these regions are retrieved and searched to find the records that match the query. In order to minimize number of disk accesses, each bucket that matches the query is retrieved exactly once.

CHAPTER IV

BANG FILE

Introduction

The BANG file[7], a balanced and nested grid file, is a multidimensional file structure that is well suited for managing dynamic databases. As in the grid file[14], the BANG file maintains the correspondence between the data bucket and grid region. The main disadvantage of the grid file structure is the rapid increase of the size of its directory, especially if the data distribution is not uniform. Most of these directory entries point to the same data bucket. In order to avoid this shortage, the BANG file structure maintains a one to one correspondence between a directory entry and a data bucket. Each directory entry of the BANG file is represented by a unique number pair $\overset{r,l}{<r,l>}$, where r is the region number and l is the level number. To achieve this compact directory size, the BANG file structure makes some modifications to the fundamental strategies of the grid file.

1. BANG file allows nested block regions. If two block regions into which the data space has been partitioned intersect, then one of these regions comple-

tely encloses the other.

2. If a data bucket overflows, the splitting is performed until the best balanced condition is achieved. One advantage of this approach is the better utilization of the secondary storage.

Figures 16 and 17 show the differences between these two file systems in handling the data. In case of overflow, the grid file stops the partitioning of the grid region as soon as the overflow condition is removed. This strategy may cause one of the resulting buckets to have a low population and the other to have high population. A few insertions or deletions into these buckets may invoke another splitting or merging algorithm. If the distribution of data is not uniform over the data space, several partitionings may be needed to remove the overflow condition. These partitionings cause the directory of the grid file structure to grow with exponential rates. In the BANG file structure, an overflow is handled by successive divisions of the overflow region until a balance in the storage utilization is achieved without significantly increasing the directory size.

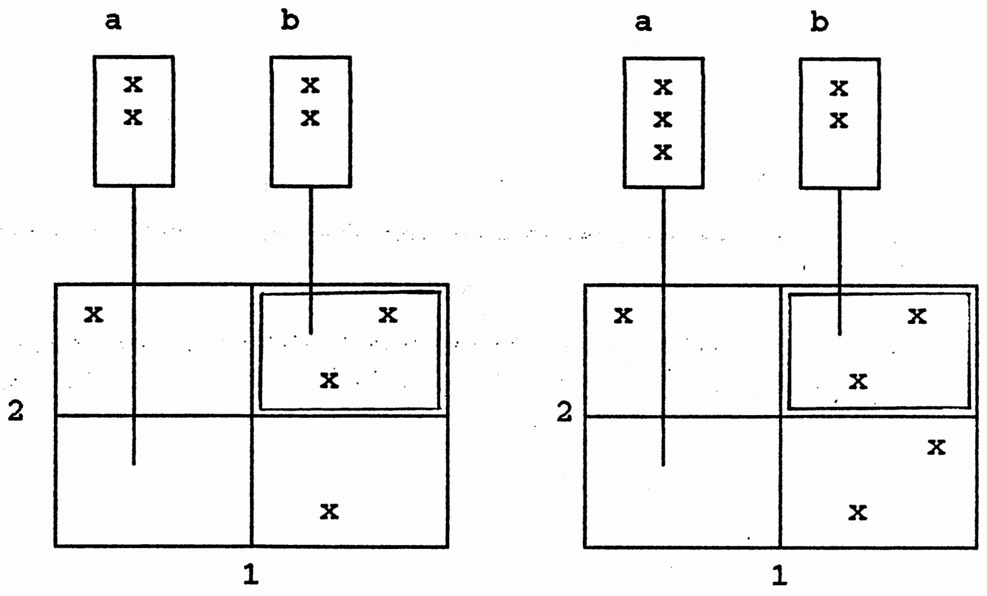


Figure 16. The BANG File Structure.
(Max. block capacity = 3)

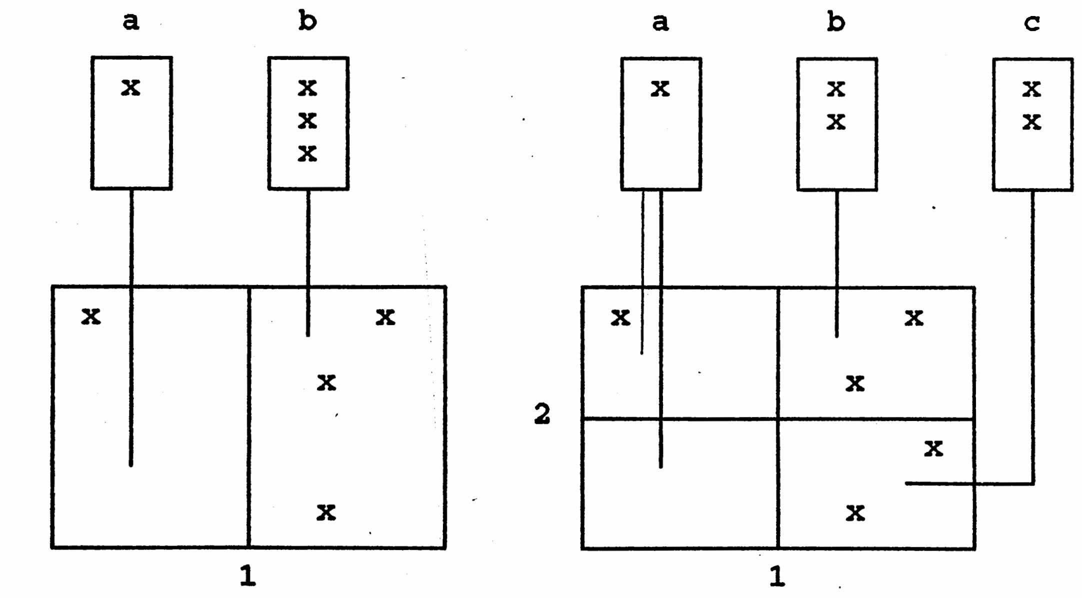


Figure 17. The Grid File Structure.

Mapping function

In this section we discuss a region numbering method that provides the following properties.

1. Each region is identified by a unique number pair <r,l>, where r represents a region number and l is the level number.
2. The region number can be easily calculated from a given set of n key values (k_1, k_2,..., k_n) and the partial level of each dimension (l_1, l_2,..., l_n), where level number $l = \Sigma l_i$.
3. Given a region number r at level l, then the region R that encloses region r at level l-1 can be easily computed.

Figure 18 shows the example of this numbering method. For the sake of simplicity we present an example of a file system with dimension = 2. Assuming that there is no preferred attribute, then the partition algorithm chooses the partition dimension in cyclic sequence. In order to show the relation between the region number and its corresponding coordinates, all region and coordinate numbers are given in binary representation.

Based on Figure 18, we derive some conclusions.

1. The total number of the grid regions at level l is 2^l. If the single grid region at level l=0 is

2^l

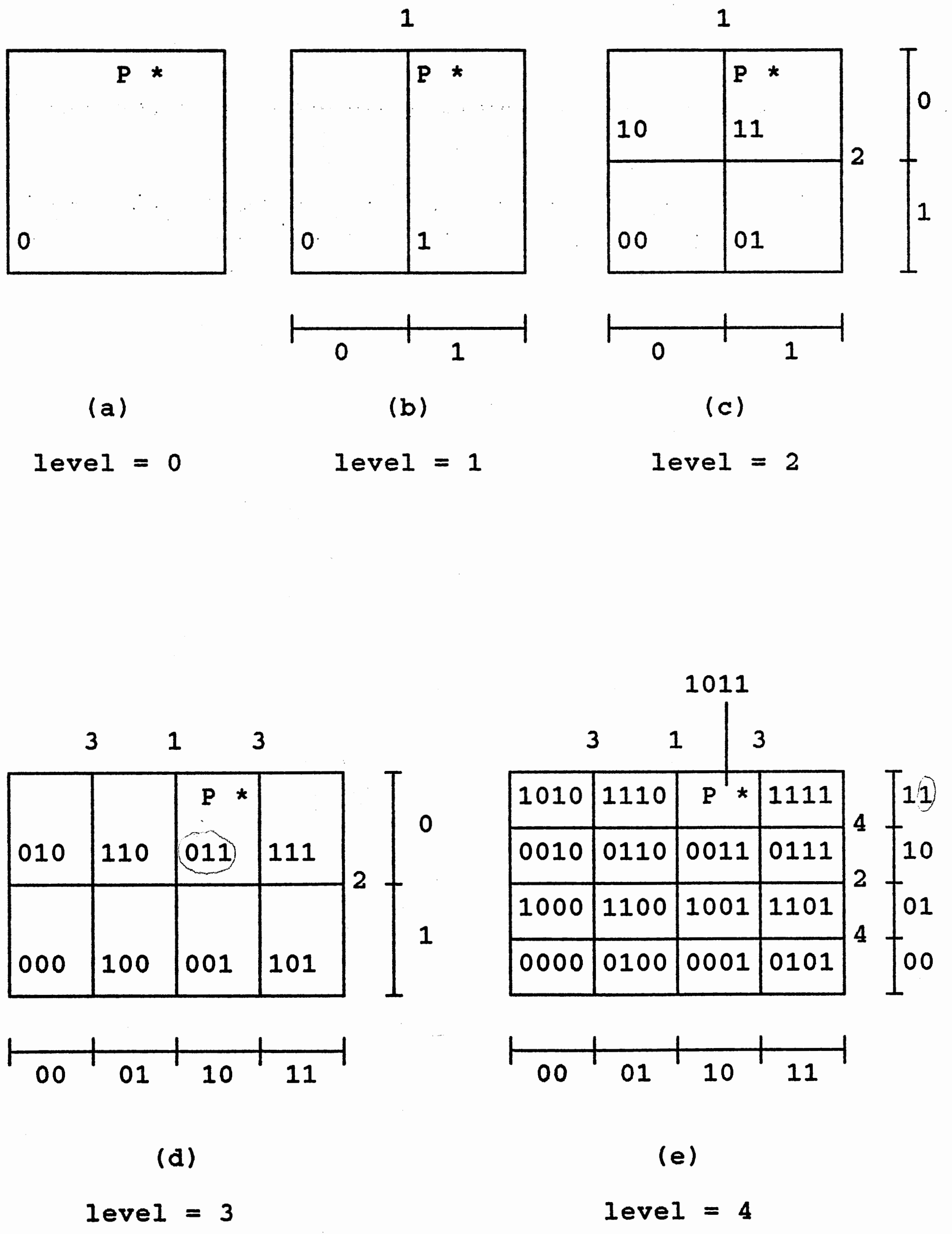


Figure 18. Numbering Method.

assigned a number r = 0, then every grid region number at level l, for all l > 0, can be represented by l bits.

2. If the dimension i is split, the domain of coordinate i is doubled by extending the the binary representation of the coordinate i by one bit(the least significant bit). As an example, the coordinate 0 (binary) of dimension 1 in Figure 18.c is split into coordinates 00 and 01 as shown in Figure 18.d.
3. If a region r at level l is split by partitioning the dimension i, then the result is 2 regions that are uniquely numbered as r and r + 2^l. This can be achieved by concatenating the least significant bit of the newly-formed coordinate in dimension i at level l+1 to the most significant bit of the corresponding region number at level l. For example, a splitting of the grid region 011 in Figure 18.d produces 2 grid regions numbered 0011 and 1011.
4. If a record p falls into a region r at level l, then the region that encloses p at level l-1 can be computed by removing the most significant bit from r. For example, a record P in Figure 18.e that is enclosed in region 1011 at level l=4 is also enclosed in region 011 in level l=3.

Figure 19 depicts a tree representation of the splitting history of the data space for dimension = 3. The purposes of this figure are to show:

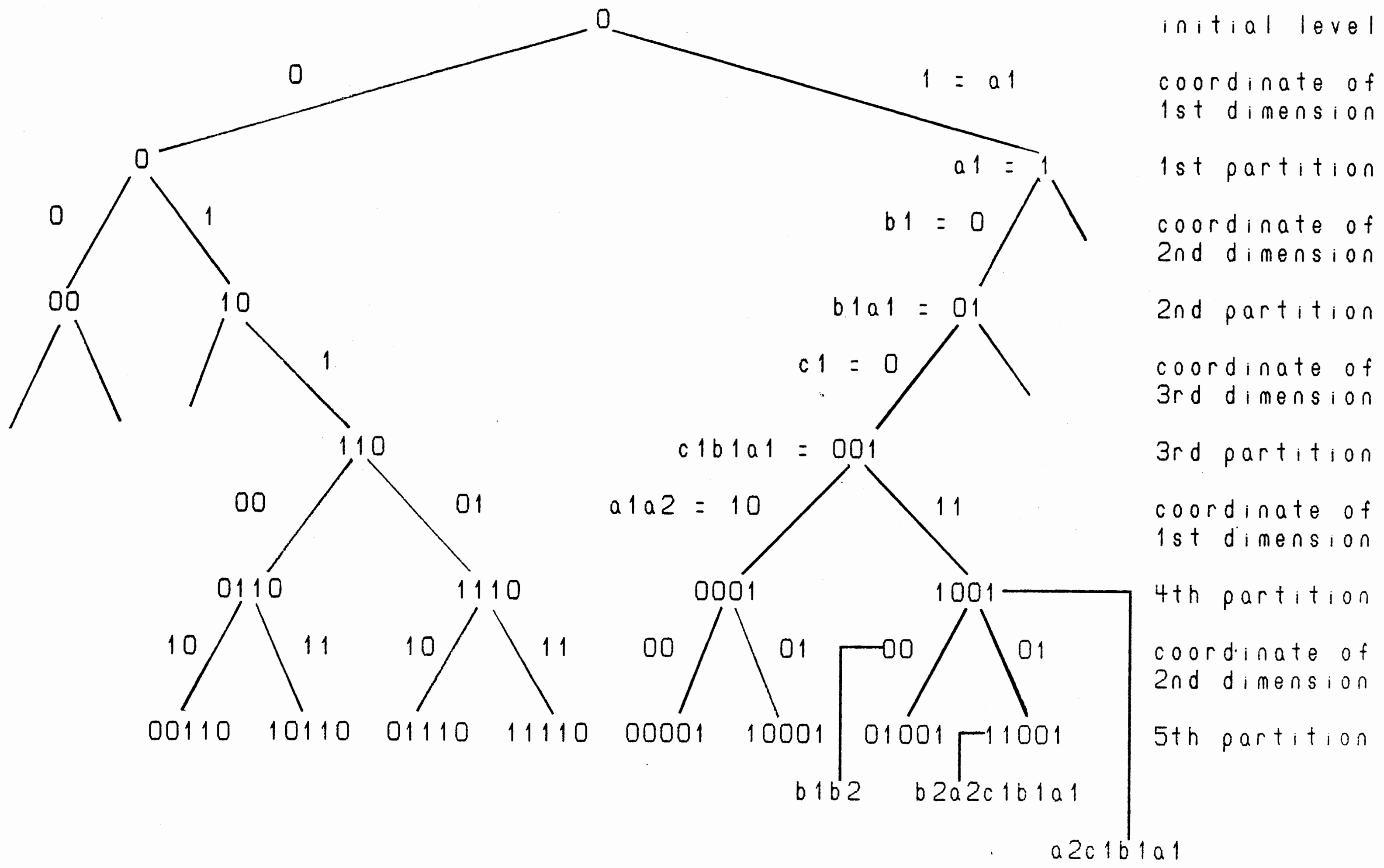


Figure 19. Splitting History for 3 Dimensional Data Space

1. the numbering method discussed above can be applied in the file system with dimension greater than 3;
2. the method to generate a region number based on the coordinate numbers.

Consider a data space with domains (D_1, D_2, ..., D_n) and the partial levels (l_1, l_2, ..., l_n) corresponding to each dimension. The smallest region number that encloses a data point P with key values (k_1, k_2, ..., k_n) can be obtained by the following steps.

1. Transform the set of key values of P into the set of coordinates (d_1, d_2, ..., d_n) of the region that encloses P.

   $f_i = k_i \;/\; |D|$ . . . . .Eq. 4.1

   f_i is the fraction of domain D_i where the key value k_i is located.

   $d_i = \lfloor f_i * 2^{l_i} \rfloor$ . . . . .Eq. 4.2

   d_i is the coordinate of dimension i at partial level l_i where the key value k_i is located.
2. Use the following algorithm to calculate the smallest region number that encloses P.

```
/* Algorithm 1 */

r = 0 ;                /* r is region number */
for(j = level; j >= 1; j--) {
   i = ((j-1) mod n);        /* n is dimension    */
   r = r*2 + (d_i mod 2);  /* d_i is coordinate  */
   d_i = d_i div 2;
}
return(r);
```

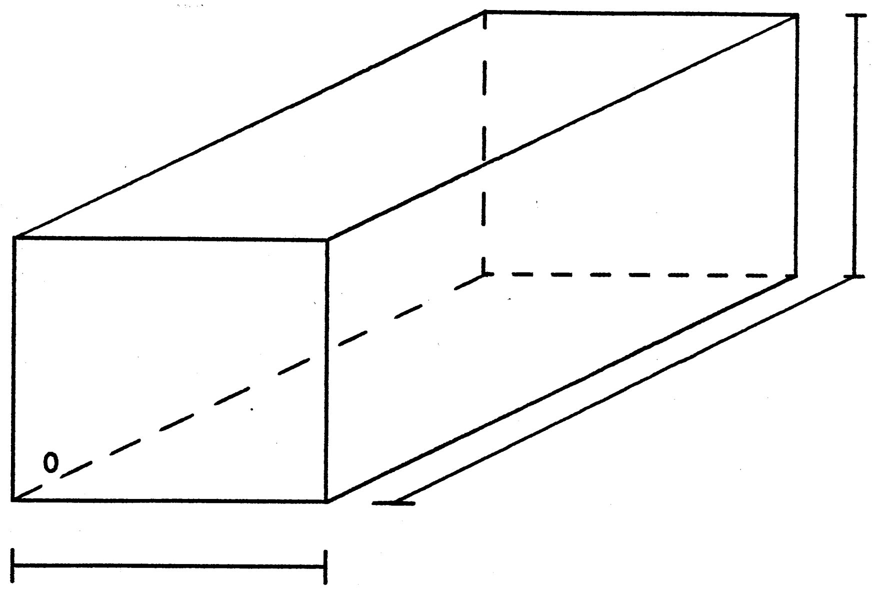
In the following we present the splitting of a three dimensional BANG file. The dimension for splitting is chosen in cyclic sequence starting with dimension 1. Figure 20.a shows the initial state of the data space. The single region at this level is numbered as region 0 and the level is 0. So, the unique number pair for this region is <0,0>, where the first 0 represents the region number and the second 0 represents the level number.

If region r at level l is split, then the two newly formed regions are numbered as r and r + 2^l, and the level is increased by one. In Figure 20.b we show the state of the data space after the first splitting on dimension 1. The region <0,0> is partitioned into region <0,1> and <1,1>. Since the region <0,1> and region <1,1> are enclosed in region <0,0>, a record that is enclosed in region <0,1> or region <1,1> is also enclosed in region <0,0>. Given a region number at current level, then all regions that enclose it at previous levels can be calculated by the following algorithm.

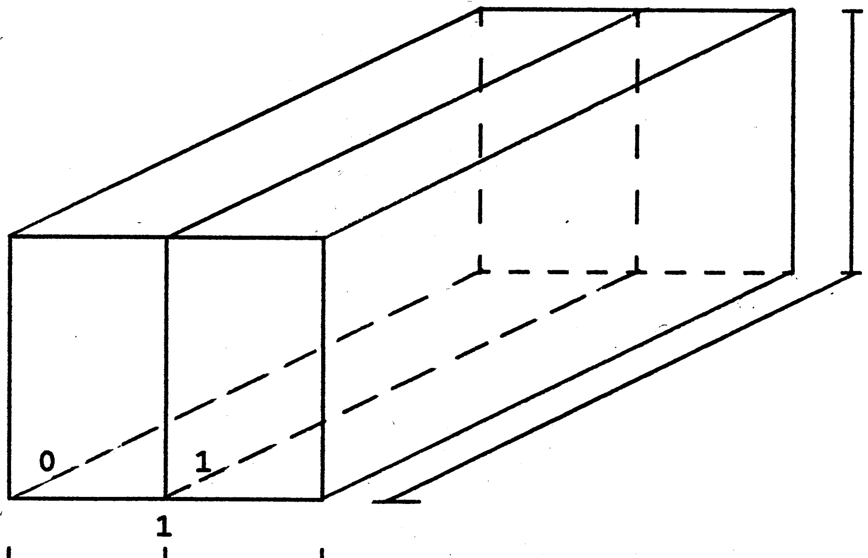
```
/* Algorithm 2 */

/* l is the current level and r is the region number
   at current level */
/* r[i] is the region number at level i */

temp = r;
for(i = l - 1; i >= 0; i--) {
   if(temp >= 2^i )
      temp = temp - 2^i ;
   r[i] = temp;
}
```

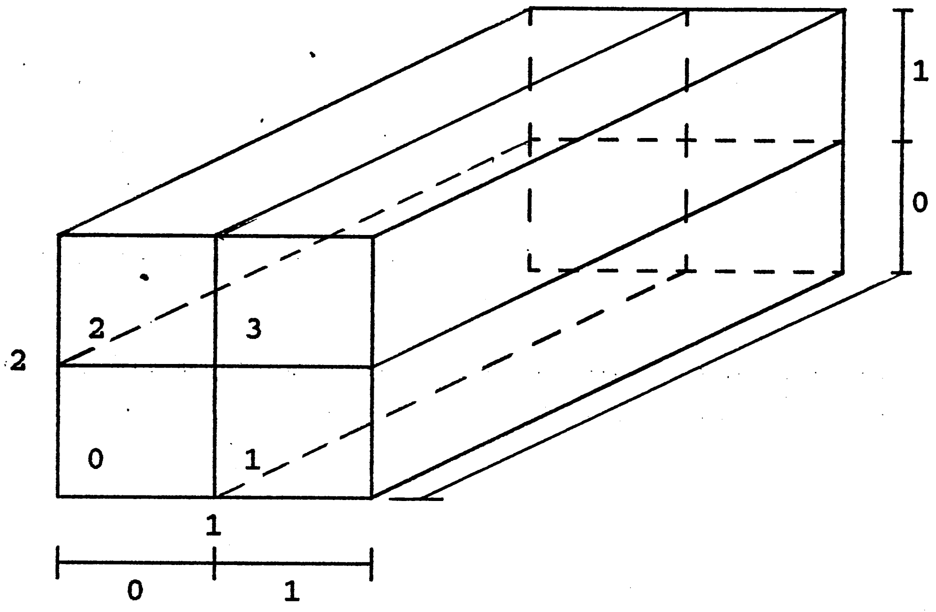


(a). Initial.

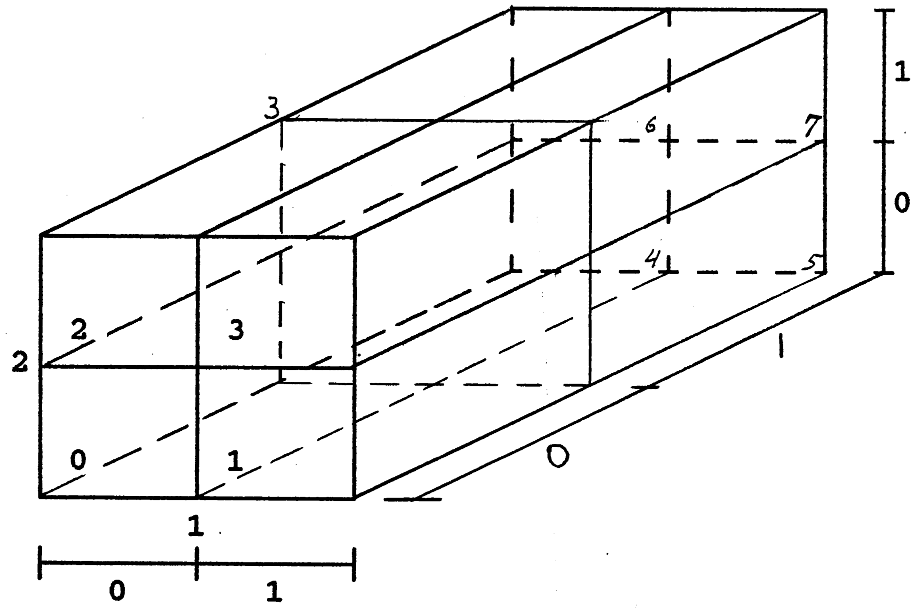


(b). 1st Partition.

Figure 20. Space Partition.

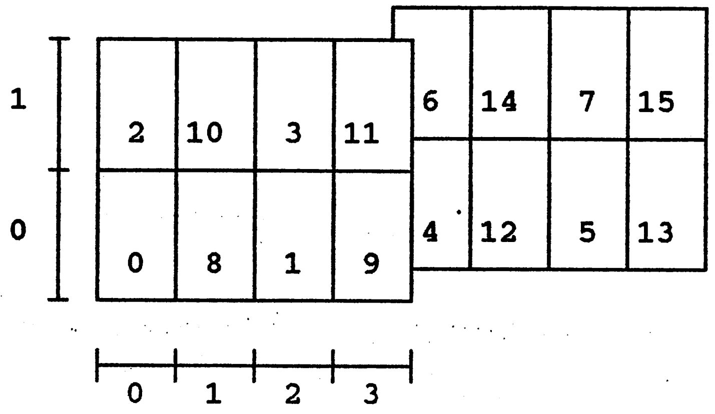


(c). 2nd Partition.

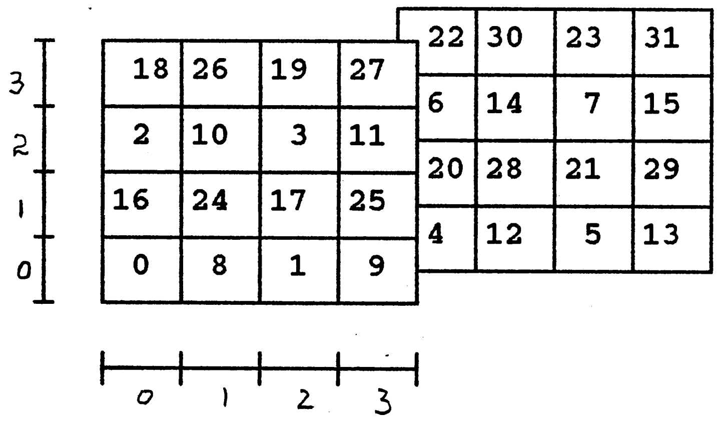


(d). 3rd Partition.

Figure 20 (cont)

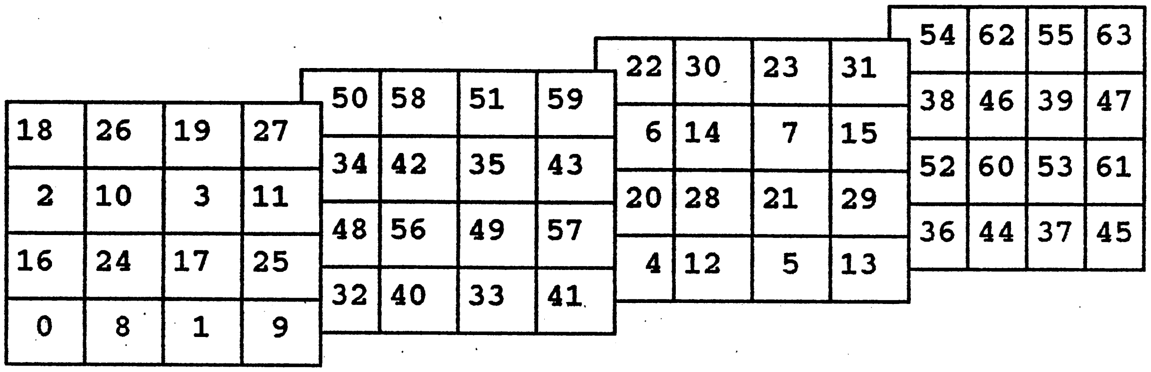


(e). 4th Partition.

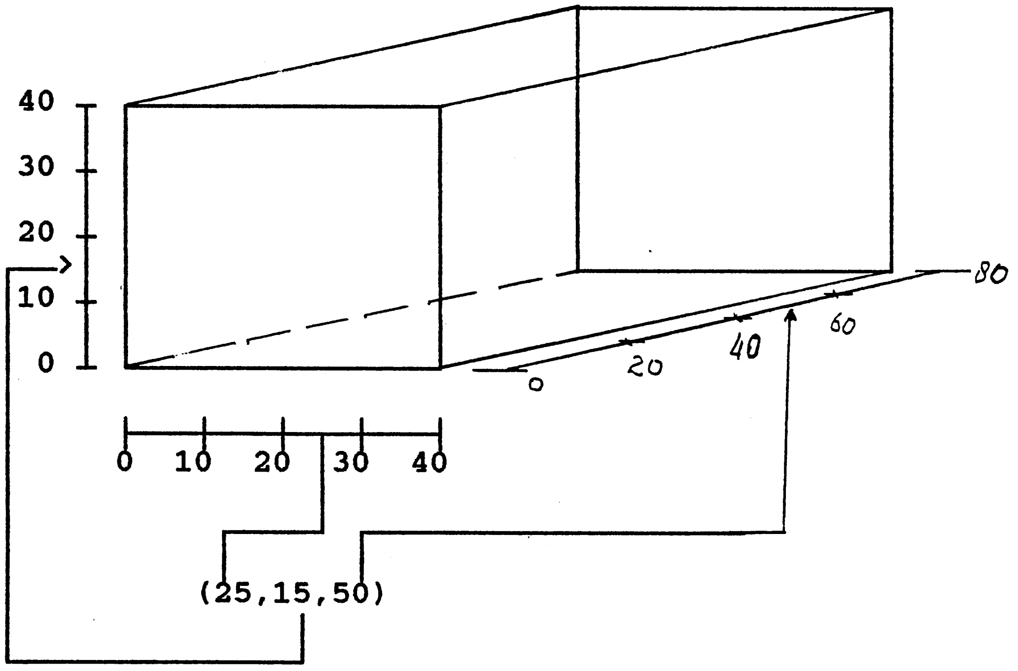


(f). 5th Partition.

Figure 20 (cont)



(g). 6th Partition.



(h). An Example of Calculating Region Number.

Figure 20 (cont)

Figure 20.h shows an example to calculate a region number. Given a record P with key values (25,15,50), then the region in which it falls can be calculated as follows.

1. Calculate the index for each dimension by using Equations 4.1 and 4.2.

   l1 = 2, f1 = 25 / 40, d1 = 2

   l2 = 2, f2 = 15 / 40, d2 = 1

   l3 = 2, f3 = 50 / 80, d3 = 2

2. By using the Algorithm 1, the region <21,6> where P lies can be calculated.

The example above shows that a region number can be obtained without having to refer to the scales. Therefore no scales are needed in the directory.

Figure 21 shows how the regions are numbered during the splitting. The predecessor of a node represents the region that encloses it at previous level. For example, the sequence of the regions that enclose region <51,6> starting from the lowest level to the highest level are: <19,5>, <3,4>, <3,3>, <3,2>, <1,1>, <0,0>. The same sequence can also be obtained by using Algorithm 2.

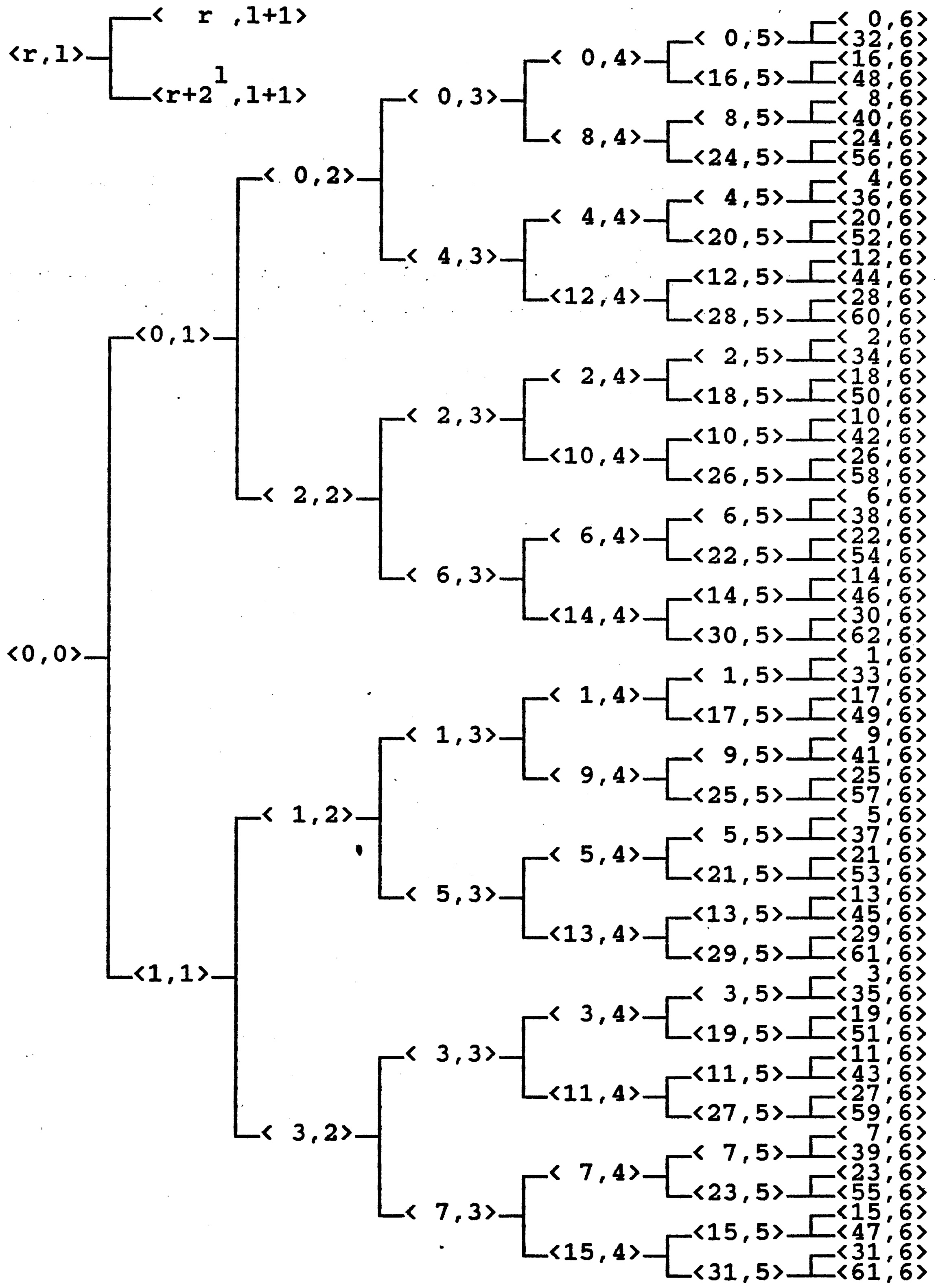


Figure 21. Region Numbering.

Directory

Each directory entry of the BANG file structure is a number pair <r,l>, where r is a unique region identifier and l is a level number. To avoid high pointer overhead as experienced by the grid file, the BANG file system maintains a one to one correspondence between the directory entry and the data block. In order to maintain it, the BANG file system does not require the subspace corresponding to a data block to be a hyperrectangle.

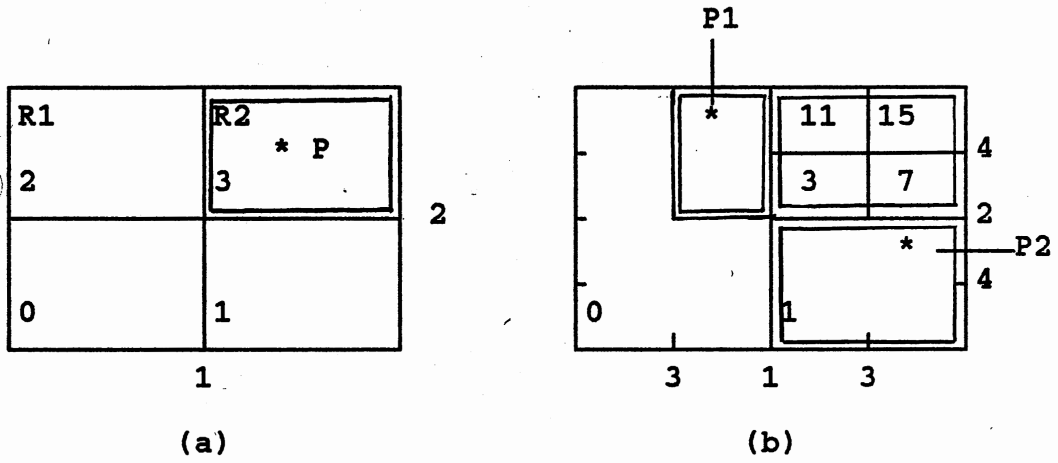


Figure 22. Nested Directory.

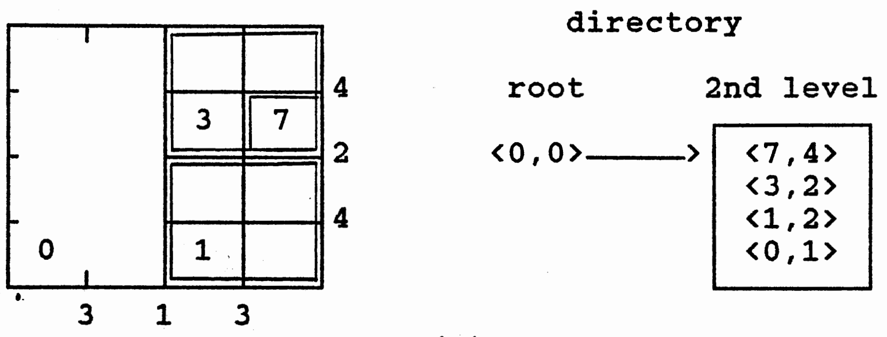
Consider the current state of a data space organized by the BANG file system as shown in Figure 22.a. The data space is partitioned into two block regions R1 and R2. R1 encloses the entire data space and R2 is enclosed or nested within

R1. S2 is the subspace enclosed by region R2, and S1 the subspace enclosed by region R1 minus subspace S2. The directory of the BANG file in this state contains 2 entries. The first entry <3,2> points to a data bucket that contains all records that fall into subspace S2, and second entry <0,0> points to a data bucket that contains all records that fall into the subspace S1. Since the data point P in Figure 22.a is enclosed by the region <3,2>, region <1,1>, and also region <0,0>, an ambiguity may arise during the search of the directory to find an entry that points to the correct data bucket which contains P.

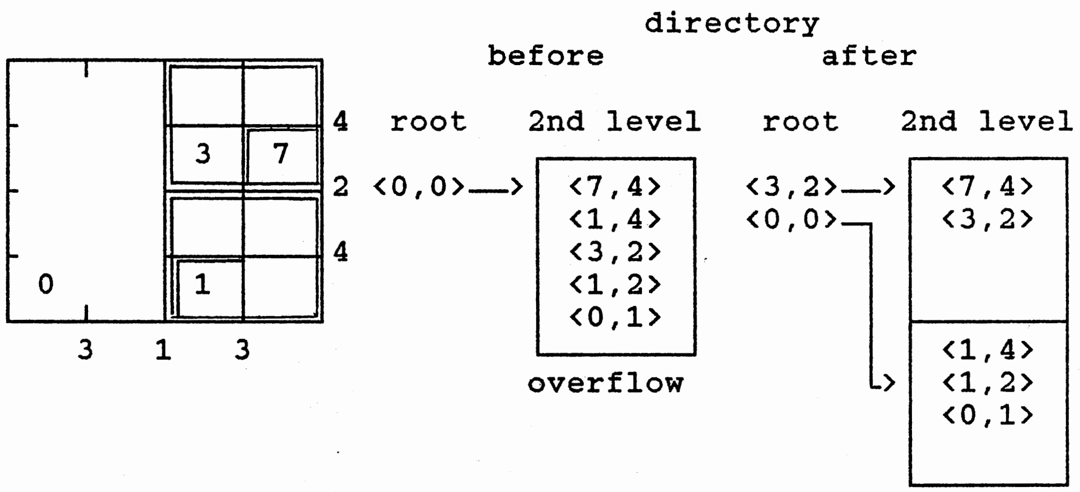
For example, a directory of the BANG file with the current state as shown in Figure 22.b has 4 entries (<0,1>, <1,2>, <3,2>, <7,3>). Suppose the directory entries are arranged as above, and a query for a record P1 is issued. The smallest region at current level 1 that encloses the location of P1 in the data space can be computed by transforming the set of its key values into a unique region identifier by using the mapping function discussed in the previous section. Based on this identifier, all regions that may enclose P1 at every level can be derived. In this case they are <7,3>, <2,2>, <0,1>, and <0,0>. During the search of directory, the first entry that matches with these regions is <0,1>, but the record P1 is not located in the data bucket pointed by it. To avoid this ambiguity, the directory entries of the BANG file structure are arranged in order of increasing partition level. During a search for a data point P , the directory is scanned to find the first smallest region that encloses P. In this example the directory entries are arranged in the order of <7,3>,<3,2>, <1,2>, <0,1>.

If the directory does not contain the entry for the smallest region r at current level l that encloses a data point P, the search is continued until the next smallest region that encloses P is encountered. As an example, the smallest region that encloses P2 as shown in Figure 22.b is <13,4>. This region identifier is not directly recorded in the directory. The search is continued for the next smallest region <5,3> that encloses P. This procedure continues until the entry <1,2> is found. This example shows that, although the directory does not contain an entry for the smallest region that encloses a data point P at current level, there is no ambiguity in locating the correct directory entry as long as the entries are arranged in increasing level number and some merging constraints are maintained.

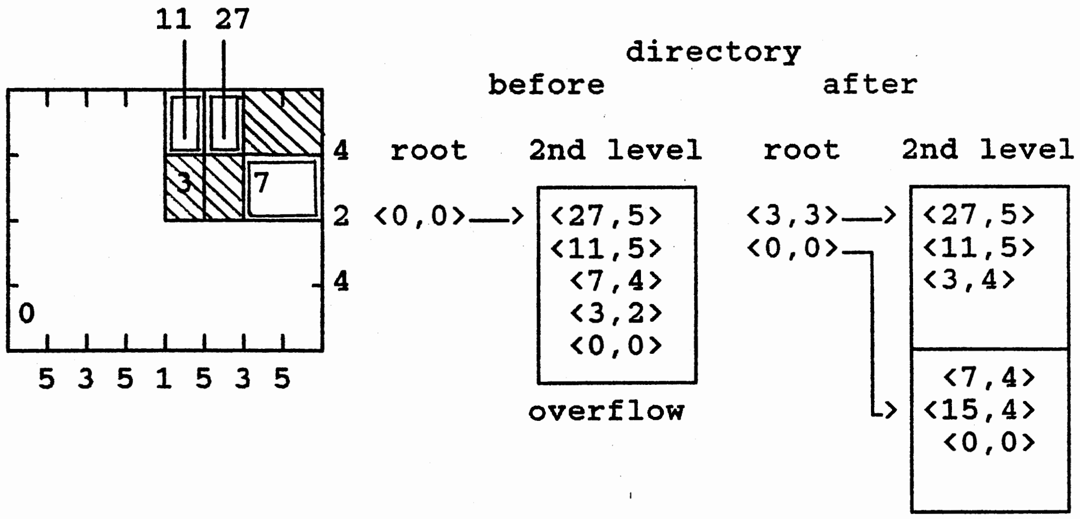
In a large database system, the size of the BANG file directory is too large to be stored in the primary memory. Therefore, the directory entries should be distributed among a sequence of disk blocks. Without a proper organization, a costly search is required to find a directory entry, i.e. approximately half of these blocks have to be searched before the correct entry can be found. One way to solve this problem is to manage these directory buckets with another BANG file. In the following discussion, we call the direc-



(a)



(b)



(c)

Figure 23. Double Level Directory.

tory of this BANG file as root directory and the directory that manages the data points as second level directory. When a directory bucket overflows, it is split by using the same method applied to the data bucket. Each directory entry is treated as a data point. A partition algorithm is invoked recursively until the best balance condition is achieved.

In the root directory, each entry points to a directory bucket. All entries of the second level directory, in which corresponding regions are enclosed in the region represented by the entry in root directory, have to be stored in the same directory bucket. The arrangement of the entries of the root directory is the same with the second level directory, i.e. all entries are arranged in increasing partition level.

In Figure 23 we show an example of how the directory bucket is split and its effects to the root directory. Assume that the maximum capacity of a directory bucket is 4 entries, and the current state of the BANG file system is shown is Figure 23.a. At first, the root directory contains a single entry which represents the whole data space. Suppose an insertion of a data point P causes a data bucket to overflow. The partitioning of this data bucket introduces a new directory entry into the corresponding directory bucket, which in turn causes this directory bucket to overflow. The result of the partitioning of this directory bucket is shown in Figure 23.b. The best balance condition is achieved after 2 partitions. All second level directory entries with corresponding regions are enclosed in region

<3,2> are located in the same directory bucket and the rest of the entries are located in the other directory bucket, which is pointed to by the root directory entry <0,0>.

Figure 23.c shows a possibility that may arise during the partitioning of an overflow directory bucket. In order to balance the contents of the resulting directory buckets, the overflow bucket is successively partitioned until l = 3. The resulting root directory entries are <3,3> and <0,0>. The assignment of the entries of the old directory bucket to the new directory buckets pointed by <3,3> and <0,0> is the same with the previous example, except for the entry <3,2> which part of its region is enclosed in region <3,3> (shaded area in Figure 23) and the other part is enclosed in region <0,0>. In this case, the region represented by <3,2> is partitioned into two regions. One of these regions (<3,4>) is enclosed in region <3,3> and the other (<15,4>) is enclosed in region <0,0>. If the content of the regions falls below a certain threshold, then the merging algorithm is invoked to maintain a reasonable storage utilization.

Update

In a dynamic file system, insertions and deletions of records are intermixed with the queries. If an insertion of a record causes the bucket to overflow, the partitioning algorithm is invoked to partition the corresponding block region into two new regions. This procedure is repeated until the best balance condition is achieved. In order to

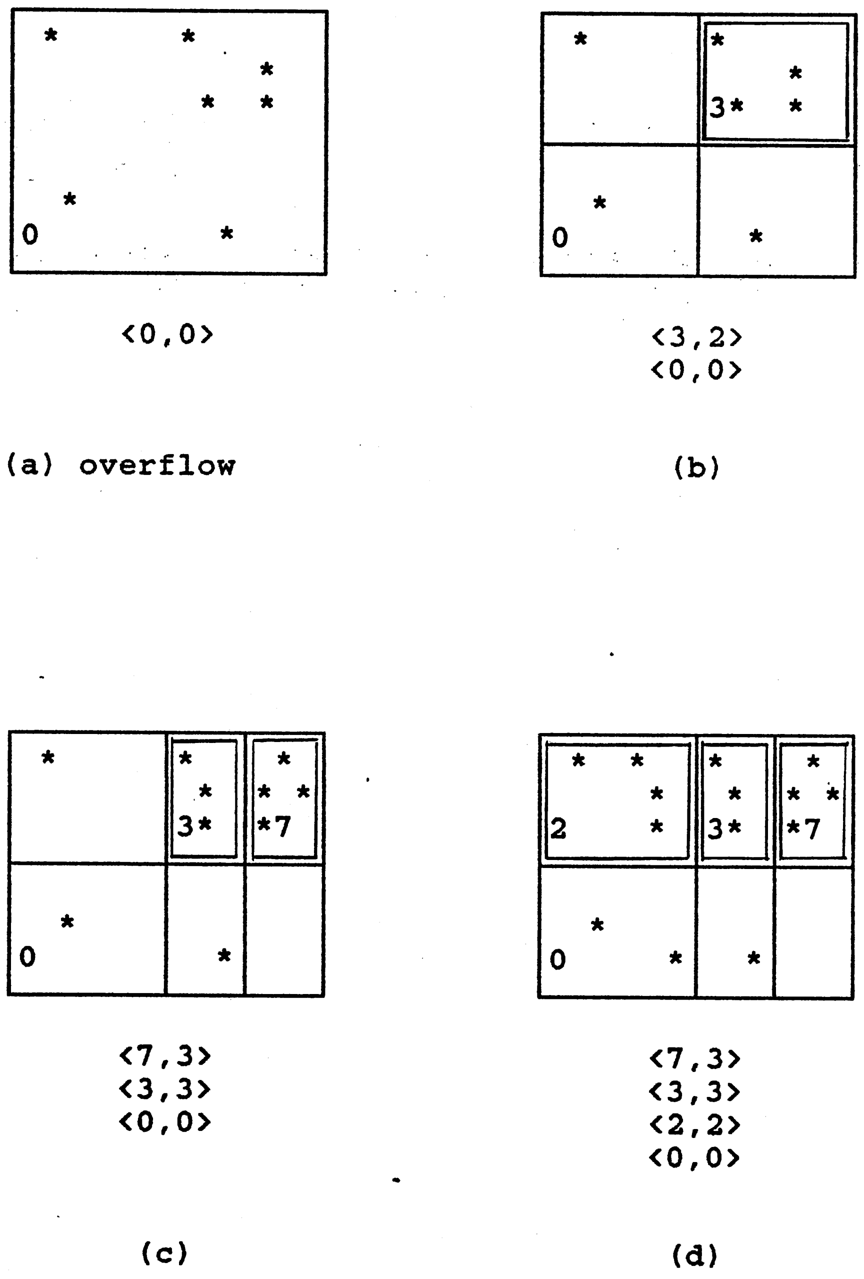


Figure 24. Splitting (Max capacity = 6).

maintain a reasonable storage utilization, a merging algorithm is invoked if a deletion causes the bucket capacity to fall below a certain threshold.

splitting

An insertion of a record into a data bucket may cause it to overflow. In order to maintain a one to one correspondence between data bucket and directory entry, the overflowing data bucket has to be partitioned. Figure 24 shows some conditions of splitting and their effects on the directory structure.

1. The partitioning of the data space is performed until the best balance condition is achieved. If n, where n > 1, partitions are needed to achieve this balance, then the directory entry <r,l> of the old region is modified into two new entries <r,l> and <r1,l+n>, where r1 is the identifier of the newly formed region. A partition of region <0,0> in Figure 24.a modifies the directory entry <0,0> into two new entries <3,2> and <0,0> as shown in Figure 24.b.
2. If the balance condition is achieved at the first level of division, then the directory entry <r,l> of the old region is replaced by the identifiers of the two resulting regions <r,l+1> and <r+2^l,l+1>. For

example, the directory entry <3,2> in Figure 24.b is replaced by directory entries <3,3> and <7,3> in Figure 24.c. In this case, the two resulting regions are called the buddy regions.

3. The partitioning of a region is always treated as a continuation of a higher level split. For example, If the data bucket pointed by grid region <0,0> in Figure 24.c overflows, the partitioning is started from level 0. The resulting regions after achieving the balance condition are identified by <2,2> and <0,0>. These two entries are added into directory to replace the entry <0,0> as shown in Figure 24.d.

Merging

To maintain reasonable storage utilization, a merging algorithm is invoked if deletion of a record causes the population of a data bucket to fall below a certain threshold. In order to avoid having this data bucket overflow soon after a few subsequent insertions, the merging is performed only if the population of the resulting data bucket does not exceed some predefined value. Figure 25 shows the strategy of choosing the victims for merging.

1. If the population of a region falls below a certain limit, the first attempt is to merge it with one of the regions which it "immediately" encloses, starting from the smallest. Suppose after a deletion, the population of the region identified by <1,2> falls

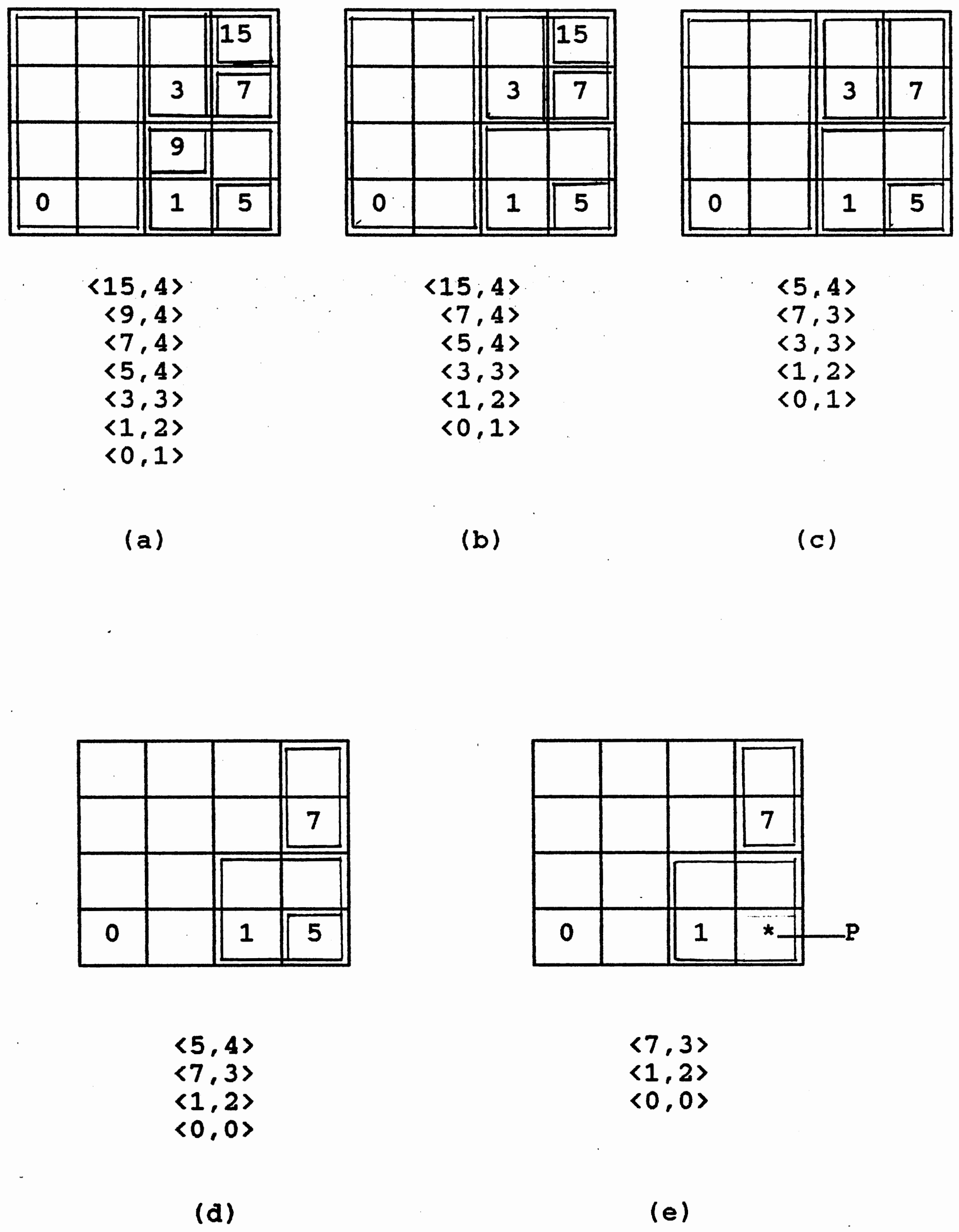


Figure 25: Merging.

below a certain limit, then the merging algorithm is invoked to merge it with the region identified by <9,4>. If it is failed, then the algorithm tries to merge it with the region identified by <5,4>.

2. If the first attempt fails, then the second attempt is to merge the region with its buddy. For example, the merging of the region <15,4> with the region <7,4>.
3. If the second attempt fails, then the third attempt is to merge a region with its "immediately" enclosing region. For example, if the contents of the region <3,3> falls below a certain limit, the merging algorithm is invoked to merge it with region <0,0>.

Since the shape of a subspace in the BANG file does not have to be a hyperrectangle, no deadlock detection is required during the merging of two regions. The reason for merging with the "immediate" enclosed or enclosing region is to avoid the ambiguity that may arise. Suppose the region <5,4> is merged with region <0,0>, which does not immediately enclose region <5,4>, then ambiguity may arise during the search for data point P as shown in Figure 25.e. When a query for the data point P is issued, the directory is searched for the entry <5,4>, the search is continued until the smallest region <1,2> that encloses data point P is found. Then the data bucket pointed by <1,2> is searched while the data point P is stored in the data bucket pointed by <0,0>.

Searching strategy

In this section we discuss 3 types of query.

1. Exact match query, one which specifies a value for each of the indexed attributes.
2. Partial match query, which specifies values for $d < n$ of the indexed attributes.
3. Region query, one in which a range is specified for each indexed attribute.

Exact match query

The general form of the exact match query is (A1 = k1), (A2 = k2), (A3 = k3), ..., (An = kn), where ki is a value of attribute Ai and n is the number of indexed attributes. In order to retrieve the correct data bucket that contains the record P, the given key values are transformed into an identifier of the smallest region <r,l> that encloses P at current level l. Based on this identifier, the record P is searched in the following steps.

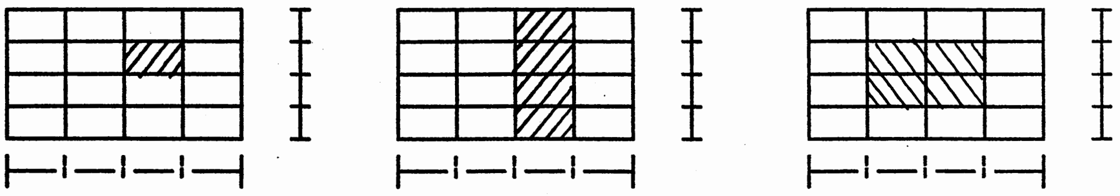
1. Search the root directory to find an entry <r1,l1> which represents the smallest region in the root directory that encloses region <r,l>.

2. Retrieve the directory bucket pointed by the entry <r1,l1>.
3. Search the directory bucket to find an entry <r2,l2> which represents the smallest region that encloses region <r,l>.
4. Retrieve the data bucket pointed by <r2,l2>. If the record P exists, then it should be located in this data bucket.

In the BANG file system, the unsuccessful and successful exact match queries can always be completed with only two disk accesses. Figure 26.a shows a simple example of the exact match query.

Partial match query

The general form of the partial match query is $(A_{i1} = k_{i1})$, $(A_{i2} = K_{i2})$, ..., $(A_{id} = K_{id})$, where $i1 < i2 < \ldots < id$, and $id < n$ is the number of specified attributes. The



(a) Exact Match Query (b) Partial Match Query (c) Region Query

Figure 26: Query

method to retrieve the data points that match the query is similar to the exact match query, except that the given key values are transformed into a set of region identifiers by the following method.

1. Transform the key values of specified dimensions into the coordinate numbers C_{i1}, C_{i2}, ..., C_{id} (refers to equations 4.1 and 4.2).
2. Calculate all region identifiers based on the combination of this coordinates with each coordinate of each unspecified dimension.

Figure 26.b shows an example of partial match query for 2 dimensions data space. The searching method of the partial match query follows.

1. Find the set of region identifiers that matches the query as shown as a shaded area in Figure 26.b.
2. Remove one region identifier from the set and use the same method as in exact match query to locate the data bucket corresponding to it.
3. Find all records in the data bucket that match the query.
4. If the set is not empty then goto 2, else the query is completed.

In order to minimize the number of disk accesses, the same directory and data buckets are loaded from disk into memory exactly once.

Region query

The general form of the region query is Li <= Ai <= Ui, where Li and Ui are the bounds for attribute Ai. The method of retrieving the matching data points in region query is exactly the same with the partial match query. Figure 26.c shows an example of region query.

CHAPTER V

PERFORMANCE COMPARISON

Implementation of 3D grid and 3D BANG file

In this chapter we discuss the algorithms of the grid and BANG file structures. These algorithms are designed to handle n dimensional records. In order to maximize the utilization of the main memory, a double level directory method is used. Since the contents of a databucket is fixed, an insertion or deletion of a record may invoke a splitting or a merging procedure, repectively. The splitting operation is performed until the overflow condition is removed and the merging operation is done until no buckets can be merged without violating the merging restriction.

In our implementation, a record consists of several keys of type integer and some additional information which is not of interest to our discussion here. For n dimensional files, each record is identified by the combination of n keys. In this project, we allow several records to have the same combination of keys. The only restriction is that the number of these records may not exceed the maximum capacity of a databucket.

Algorithm 3 shows the main body of the programs for grid and BANG file structures. Algorithm 4 is the algorithm ACTION() for the grid file structure and Algorithm 5 is for the BANG file structure.

Algorithm 3

```
main()
{
   choose the action to be done;
   if(choice is build) {
      Initialize new file information;
      ACTION(insert);
   }
   else {
      Read existing file information;
      if(choice is range query)
         Performs range query;
      else
         ACTION(choice);
   }
}
```

Algorithm 4

```
ACTION(choice)
{
   for(all records) {
      Get root region;
      Read subdirectory bucket;
      Get subdirectory region;
      Read databucket;
      if(action is insertion) {
         If(databucket is not full)
            Insert the record into the bucket;
         else
            Split databucket and insert the record into
                                        one of the buckets;
      }
      else if(action is exact match query or deletion)
         for(all records in the bucket){
            Check if there is a record that match the
                                             input record;
            if(yes) {
               if(choice is exact match query)
                  Print out the result;
               else if(choice is deletion)
                  Delete the record from the file;
            }
         }
      }
   }
}
```

Algorithm 5

```
ACTION(choice)
{
   for(all records) {
      Find smallest region r that encloses the record;
      Find the smallest entry r1 in the root directory
                                      that encloses r;
      Read subdirectory bucket pointed by r1;
      Find the smallest entry r2 in the sub directory
                                      that encloses r;
      Read databucket pointed by r2;
      if(action is insertion) {
         If(databucket is not full)
            Insert record into the bucket;
         else
            Split databucket and insert the record into
                          one of the resulting buckets;
      }
      else if(action is exact match query or deletion)
         for(all records in the bucket){
            Check if the record match the input record;
            if(match) {
               if(choice is exact match query)
                  Print out the result;
               else if(choice is deletion)
                  Delete the record from the file;
            }
         }
      }
   }
}
```

Grid file algorithms

In our implementation of the grid and BANG file structures, the root directory is stored in the main memory and the subdirectory is stored in a directory file.

The data structures used to describe the root and subdirectory are given by the following structure declarations:

```
ROOT_DIRECTORY {
    int     bound_number[DIMENSION];
    int     *scales[DIMENSION];
    CELL    *cell;
}
```

```
SUBDIRECTORY {
    int     bound_number[DIMENSION];
    int     scales[DIMENSION][MAX_BOUND_PER_DIMENSION];
    CELL    cell[MAXIMUM_CELL_PER_BUCKET]
    int     lowest[DIMENSION];
}
```

The field identifiers in the data structures for root and subdirectory have the following meaning:

field identifier	description
bound_number	Represents the number of the scale boundaries for each dimension.
scales	Stores the grid scale for each dimension. Each scale is a one dimension array.
cell	Grid directory array which is stored in row major order. Each entry of this array consists the pointer to a bucket

	and the contents of the bucket. The size of 'scales' and 'cell' are fixed for subdirectory and dynamically allocated for the root directory.
lowest	Lowest key of each dimension for a subdirectory.

An insertion or deletion of a record may cause a databucket to overflow or underflow. In our implementation of the grid file, the overflowing databucket is split by using the "binary buddy method". The splitting dimension is chosen based on the 'level' value of each dimension of the region to be split. The level value represents the number of splitting operations done to obtain the interval of the region at the corresponding dimension. Algorithm 6 shows the method to calculate the level value and to find the splitting dimension.

Algorithm 6

```
FIND_SPLITTING DIMENSION(reg)
{
   for(i = 0; i < DIMENSION; i++) {
       range1 = MAX_KEY - MIN_KEY;
       range2 = reg->up_bound[i] - reg->low_bound[i];
       for(level = 0; range1 != range2; level++)
           range1 = range1 / 2;
       reg->level[i] = level;
   }
   for(choice = 0,i = 1; i < DIMENSION; i++)
       if(reg->level[choice] == reg->level[i])
          if(reg->share[choice] < reg->share[i])
             choice = i;
       else if(reg->level[choice] > reg->level[i])
          choice = i;
   return(choice);
}
```

The variable 'reg' of Algorithm 6 represents a region of the directory, where all array elements in the region point to the same bucket. The information of the subdirectory region that points to the databucket, where a record should be located, is stored in the variable 'gd_c' and the information of the root directory region that points to the same subdirectory bucket is stored in the variable 'gr_c'. In the following we present the data structure used to store the region information of a directory.

```
REGION_INFORMATION {
   int      lower_bound[DIMENSION];
   int      upper_bound[DIMENSION];
   int      left_index[DIMENSION];
   int      right_index[DIMENSION];
   int      level[DIMENSION];
   int      share[DIMENSION];
   int      pointer;
   int      number;
}
```

The field identifiers of the structure REGION_INFORMATION have the following meaning:

field identifier	description
lower_bound & upper_bound	Show range of the attributes of a region. All records with key values within this range should be located in the bucket pointed by region pointer.
left_index & right_index	Show the leftmost and rightmost indices of the region.

level	Shows the splitting history of the region.
share	Shows the number of directory array elements in each dimension that have the same pointer.
pointer	Pointer to a bucket.
number	Shows the contents of the bucket pointed by the pointer.

Given a record, the following algorithms illustrate the method to find the directory region that points to the bucket where the record can be found. The variable 'type' may be ROOT or SUBDIR to indicate the root or subdirectory region. The variable rd_c is equal to gr_c if type is ROOT and equal to gd_c if type is SUBDIR. The argument cond is used during the range query to show whether the value of argument key itself is or is not included in the query region. If the range query condition is >= or <= key then the value of cond is set to 1, else if the range query condition is > or < key then the value of cond is set to 0.

Algorithm 7

```
GRID_REGION(type,scales,bound_number,cell,rd_c)
{
   GET_INDICES(key,indices,scales,bound_number,cond);
   j = CELL_NUMBER(indices,bound_number);
   for(i = 0; i <  DIMENSION; i++) {
       Find the leftmost directory element at dimension
            i that has the same pointer with element j;
       Find the rightmost directory element at dimensi-
         on i that has the same pointer with element j;
   }
}
```

Algorithm 8

```
GET_INDICES(key,indices,scales,bound_number,cond)
{
   for(i = 0; i < DIMENSION; i++) {
       index = 0;
       if(cond[i] == 1)
           while(key[i] >= scales[index])
               index = index + 1;
       else
           while(key[i] > scales[index])
               index = index + 1;
       indices[i] = index;
   }
}
```

Algorithm 9

```
CELL_NUMBER(indices,bound)
{
   cellnumber = 0;
   for(i = DIMENSION - 1; i >= 0; i--)
      cellnumber= cellnumber * bound[i] + indices[i];
   return(cellnumber);
}
```

In order to maintain the two disk access principle, a databucket splitting procedure is invoked if the insertion of a record cause a databucket to overflow.

Algorithm 10

```
DATABUCKET_SPLITTING()
{
   while(databucket overflows ){
      dim = FIND_SPLITTING_DIMENSION(gd_c);
      if(subdirectory bucket is not overflow after the
                                    databucket splitting) {
         if(the overflow condition is removed ){
            splitting_key =
                   (gd_c.lower[dim] + gd_c.upper[dim])/2;
            Split databucket into two;
            Write both buckets into data file;
            Set lptr and rptr to point to the left and
                          right regions after splitting;
         }
```

```
            else {
               Split subdirectory region without removing
                                          the overflow condition;
               Set lptr and rptr to point to the left and
                              right regions after splitting;
            }
            MODIFY_DIRECTORY_REGION(SUBDIR,dim,lptr,rptr,
                 lnum,rnum,grd_c,bnum,cells,scales);
            if(overflow condition is solved)
               Write subdirectory bucket into file;
         }
         else
            Split subdirectory bucket;
      }
   }
```

To maintain the correspondence of the subdirectory elements and the databuckets, the subdirectory region that points to the overflowing databucket has to be modified. The splitting of a databucket is continued until the overflow condition is removed, i.e., at least 1 record is moved to another bucket.

A modifying of the directory region may or may not introduce a new boundary into the scale of the splitting dimension. Figure 27 shows an example of directory region splitting. Let the shaded area represent the directory region that points to the overflowing bucket and p1, p2, p3, and p4 represent the pointers to the databucket. If the region consists of more than one element in the splitting dimension, then the splitting is done by modifying the pointers as shown in Figure 27.b. If the region consists of only one element in the splitting dimension, the splitting is done by insertion of a new boundary into the scale of the splitting dimension as shown in Figure 27.c.

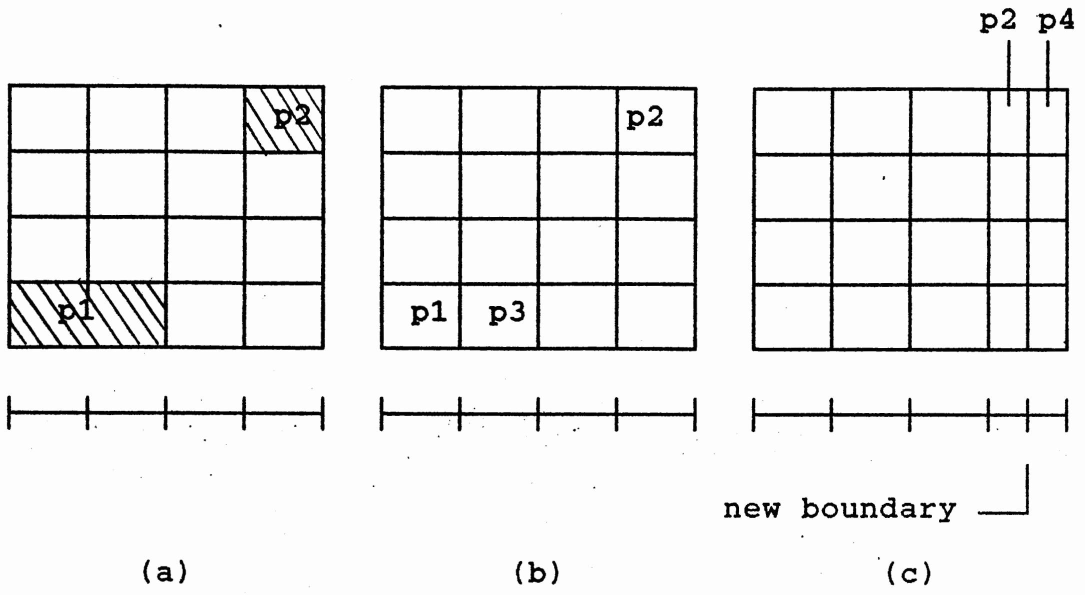


Figure 27. Splitting of Directory Region.

Algorithm 11

```
MODIFY_DIRECTORY_REGION(type,dim,lptr,rptr,lnum,rnum,
                          grd_c,bnum,cells,scales)
{
  if(number of cells in the region at splitting
                                    dimension > 1)
     MODIFY_REGION_POINTER(type,dim,lptr,rptr,lnum,
                     rnum,grd_c,bnum,cells,scales[dim);
  else
     INSERT_NEW_BOUND(type,dim,lptr,rptr,lnum,rnum,
                          rd_c,scales,cells,bnum);
     if(type == SUBDIR)
        modify the root region that points to the split
                                        subdirectory;
}
```

Algorithm 12

```
MODIFY_REGION_POINTER(type,dim,lptr,rptr,lnum,rnum,
                         region,bnum,cells,scales[dim])
{
   find the index where the region splitting occurs;
   for(i = 0; i < DIMENSION; i++)
      ptr[i] = region->low_index[i];
   for(flag = 1; flag; ) {
      number = element number with indices ptr[];
      if(element is in the right of the splitting
                                         boundary) {
         cell[number]->ptr = rptr;
         cell[number]->number = rnum;
      }
      else {
         cell[number]->ptr = lptr;
         cell[number]->number = lnum;
      }
      flag = LOOP_IS_NOT_OVER(ptr,region->low_index,
                                 region->high_index);
   }
}
```

Algorithm 13

```
INSERT_NEW_BOUND(type,dim,lptr,rptr,lnum,rnum,rd_c,
                                    scales,cell,bnum)
{
   boundary = index where a new bound is inserted;
   for(all dimension){
      low[] = ptr[] = 0;
      high[] = last index of each region;
      old_bound[] = # of boundaries before splitting;
      new_bound[] = # of boundaries after splitting;
   }
   if(type == ROOT)
      get memory for new root directory;
   else
      get new subdirectory bucket;
   for(all cells in the directory ) {
      old_cell = CELL_NUMBER(ptr,old_bound);
      if(ptr[dim] < boundary){
         new_cell = CELL_NUMBER(ptr,new_bound);
         copy old cell information into new cell;
      }
      else if(ptr[dim] > boundary){
         ptr1 = the indices of the new cell;
         new_cell = CELL_NUMBER(ptr1,new_bound);
         copy old cell information into new cell;
      }
      else {
         ptr1 = the indices of the new cell;
         new_cell1 = CELL_NUMBER(ptr,new_bound);
         new_cell2 = CELL_NUMBER(ptr1,new_bound);
         if(old cell is not included in the region)
            copy old cell information into new cell1
                                       and new cell2;
            else {
               set info of new cell1 to lptr and lnum;
               set info of new cell2 to rptr and rnum;
            }
         }
         flag = LOOP_IS_NOT_OVER(ptr,low,high);
      }
      Insert a new boundary into the scale of splitting
                                                 dimension;
      Modify the boundary number;
   }
```

In order to develop the algorithms for n dimensional grid file, we use the loop

```
for(i = 0; i < n; i++)
   ptr[i] = low[i];
for(flag = 1; flag; ){
    :
    :
    :
    :
    flag = LOOP_IS_NOT_OVER(ptr,low,high);
}
```

to replace the loops

```
for(i0 = low[0]; i0 < high[0]; i0++){
   for(i1 = low[1]; i1 < high[1]; i1++){
           :
           :
        for(in = low[n]; in < high[n]; in++){
           :
           :
           :
        }
   }
}
```

Algorithm 14

```
LOOP_IS_NOT_OVER(ptr,low,high)
{
   ptr[0]++;
   for(i = 0; i < DIMENSION - 1; i++) {
      if(ptr[i] <= high[i])
         return(1);
      else {
         ptr[i] = low[i];
         ptr[i+1]++;
      }
   }
   if(ptr[DIMENSION-1] > high[DIMENSION-1])
      return(0);
   else
      return(1);
}
```

The insertion of a new boundary into the directory scale may cause the subdirectory bucket to overflow. In this condition the subdirectory bucket and the corresponding root directory region has to be split. Algorithm 15 shows the method to split the directory bucket. The argument 'mid' represents a boundary in scale of dimension 'dim' where splitting occurs.

In order to avoid the condition in which a databucket is pointed by several cells of different subdirectory buckets, before the splitting of the subdirectory bucket, all databuckets that are pointed to by a region that encloses the array elements in both sides of 'mid' have to be split.

Algorithm 15

```
SPLIT_DIRECTORY_BUCKET(bucket1,mid,bucket2,dim)
{
   Split all databuckets that are shared by the
                    grid cells in both side of 'mid';
   SPLIT_SUBDIRECTORY_TO_TWO(bucket1,mid,bucket2,dim);
}
```

Algorithm 16

```
SPLIT_SUBDIRECTORY_TO_TWO(bucket1,mid,bucket2,dim)
{
   for(all dimension) {
      lidx_num[]=number of boundary for 'left' bucket;
      ridx_num[]=number of boundary for 'right' bucket;
   }
   for(all cell){
      old_cell = cell number at old bucket;
      new_cell = cell number for new bucket;
      if(old_cell is in the left of mid)
         Copy old cell into new cell of bucket1;
      else
         Copy old cell into new cell of bucket2;
   }
   for(i = 0; i < DIMENSION; i++){
      bucket1->b_num[i] = lidx_num[i];
      bucket2->b_num[i] = ridx_num[i];
      bucket1->lowest[i] = gdr_bct.lowest[i];
      if(i != dim){
         bucket2->lowest[i] = gdr_bct.lowest[i];
         Copy all boundaries of scales[i] to bucket1
                                          and bucket2;
      }
      else {
         bucket2->lowest[i] = (gr_c.low_b[i] +
                                     gr_c.up_b[i] / 2;
         for(all boundaries)
            if(boundary is in the left of mid)
                Copy the boundary to scales of bucket1;
            else
                Copy the boundary to scales of bucket2;
      }
   }
}
```

Deletion of a record may cause the contents of the databucket to fall below a certain threshold. In order to maintain a reasonable bucket utilization, a merging procedure is invoked to attempt to merge the underflowing databucket with another databucket. Two databuckets can be merged only if all conditions discussed in the previous chapter are satisfied. The merging algorithm is shown below.

Algorithm 17

```
MERGING()
{
   for(; ;)
      if(subdirectory encloses to several databuckets)
         FIND_MERGING_CANDIDATE(SUBDIR,dim,MAX_DATA,
                                            mg_c, gd_c,total);
         if(merging candidate is found) {
            Merge databuckets;
            Modify subdirectory;
         }
         else
            return;
      }
      else
         single = 1;
      if(subdirectory is underflow) {
         if(merging candidate is found) {
            Merge subdirectory buckets;
            modify the root directory;
         }
         else
            if(single)
               return;
      }
   }
}
```

Algorithm 18

```
FIND_MERGING_CANDIDATE(type,dim,max,mg_c,gd_c,total)
{
   for(all dimension) {
      Find the buddy of the underflowing region;
      if (the contents after merging exceeds the
                                      upper threshold)
         Try another dimension;
      if(no deadlock occurs after merging)
         Merging candidate is found and return;
      else
         Try another dimension;
   }
   No merging candidate;
}
```

Deadlock is a condition where no regions can be merged because the resulting region would not have the box shape. In order to prevent deadlock, two regions are merged if the deadlock doesnot happen after the merging. The following algorithm is used to check the occurrence of a deadlock. Let R1 and R2 be the regions to be merged and R be the resulting region after the merge. Let B be a directory region and both B1 and B2 be two regions resulting from the binary splitting of the region B. B1 and B2 are two disjoint regions.

Algorithm 19

```
DEADLOCK_CHECKING()
{
   do {
      if(B == R) {
         No deadlock occurs after the merging of region
                                     R1 and R2; return;
      }
      else {
         Split B into two disjoin regions B1 and B2;
```

```
            if(splitting is not success) {
               deadlock occurs if the the regions are
                                                    merged;
               return;
            }
            else {
               if (R is include in B1)
                  B = B1;
               else
                  B = B2;
            }
         }
      }
   }
```

A directory block B is said to be deadlock free if all regions in B can be successively merged into B. Let B be a directory block, if B can be split into two disjoint regions B1 and B2. Then the merging of regions B1 and region B2 is deadlock free too. Without losing generality, let R be included in B1. If B1 can be split into two disjoint regions B3 and B4 , then the merging of region B3 and region B4 is deadlock free. If the process can be continued until the regions R1 and R2 are obtained, then the merging of the regions R1 and R2 should also be deadlock free.

A boundary of directory scale may be removed if it is no longer needed. Given an index x of a dimension i, if all directory elements with index x at dimension i have pairs with the elements with index x+1 at the same dimension, then the boundary that splits the indices x and x+1 can be removed. The removing of a boundary from a scale may cause the contents of the subdirectory bucket to fall below a certain threshold. In this condition, a procedure that attempts to merge the directory buckets is invoked.

Algorithm 20

```
MERGE_SUBDIRECTORY_BUCKET()
{
   for(i = 0; i < DIMENSION; i++){
      if (i is not the merging dimension)
         Merge the scales of two old subdirectory
                                     buckets into new scale;
      else
         Concate the scales of two old subdirectory
                                    buckets into a new scale;
   }
   Copy all elements of the subdirectory buckets into
                                           the new bucket;
   Modify the root region after merged;
}
```

In our implementation of the grid and BANG file structures, we support two types of query, the exact match query and the range query. A successful or unsuccessful exact match query can always be completed in two disk accesses. The first access is to read the directory bucket and the second access is to read the databucket.

Range query is a query where the range of values is specified for all dimensions. In range query, all buckets that intersect with the query region are retrieved. To increase the performance of a range query, a bucket that is pointed to by several directory elements is retrieved only once. The following algorithm shows the method to check whether a region has or has not been processed. The argument low_idx denotes the leftmost indices of query region for each dimension and the argument idx represents the indices of the current directory element.

Algorithm 21

```
CHECK_NEIGHBOR(current,idx,low_idx,bnum,cell)
{
   for(i = 0; i < DIMENSION; i++){
      if(idx[i] > low_idx[i])  {
         number = left neighbor of current cell;
         if(the cell[current] and cell[number] point to
                                          the same bucket) {
            The bucket has been processed;
            return(1);
      }
   }
   The bucket hasnot been processed;
   return(0);
}
```

In our implementation of the grid file structure, the range value of a cell with index i at dimension j is [$left_i$,$right_i$). Given a query with a range $left_i$ <= x < $right_i$, then i is the only index at dimension j that intersects with the query region. If the range is $left_i$ <= x <= $right_i$, then the indices i and i+1 of dimension j intersect with the query region. Algorithm 22 shows how the range query is performed. The arguments low and high represent the lowest and the highest key values of the query region for each dimension. The argument cond_r is set to 1 if the highest key is included in the query range, and set to 0 if the highest key is not included in the query region.

Algorithm 22

```
RANGE_RETRIEVAL(low,high,cond_l,cond_r)
{
   Get the leftmost and rightmost indices of the root
        directory that intersect with the query region;
        for(all directory elements in the region){
           Check if the subdirectory bucket pointed by
             current root directory cell has been processed;
           if(it has not been processed){
              Read subdirectory bucket;
              Get the leftmost and rightmost indices of the
              subdirectory that intersect with query region;
              for(all directory elements in the region){
                 Check if the databucket pointed by current
                       subdirectory cell has been processed;
                 if(it has not been processed)
                    Read databucket and retrieve all records
                                          that match the query;
              }
           }
        }
     }
```

BANG file algorithms

The main difference between the grid file and the BANG file structures is the method of splitting the overflow bucket and the merging of the underflow bucket. In the grid file structure, the splitting is stopped as soon as the overflow condition is removed. In the BANG file structure, the splitting is continued until the best balance condition is reached.

As discussed in Chapter 4, each directory entry of the BANG file structure is a unique pair of region number and level number. The data structures for the root and subdirectory are declared as follows:

```
     NODE {
        long           region;
        int            level;
        int            header;
        int            pointer;
     }
```

```
ROOT_DIRECTORY {
   int   header;
   NODE  *cell;
}



SUBDIRECTORY {
   int   header;
   NODE  cell[MAXCELL];
}
```

The field identifiers in the data structures above have the following meanings.

field identifiers	description
region	Denotes the region number of a directory entry.
level	Denotes the number of partition needed to obtained a region.
header in NODE	Shows the contents of the bucket pointed by the pointer.
pointer	Points to a bucket. For root directory it points to the subdirectory bucket and for subdirectory it points to the databucket.
header in ROOT and SUBDIRECTORY	Shows number of directory entries in root and subdirectory.

Algorithm 5 shows how to find a bucket where a record should be located. Given a record r, Let d be the databucket where record r should be located if it exists and s be the subdirectory bucket that contains the pointer to the databucket d. The smallest region R that encloses the record r can be calculated by using Algorithm 1. In order to locate the subdirectory bucket s, the root directory is searched to find the smallest directory entry that encloses region R. Given the smallest region R, Algorithm 23 shows how to find the root directory entry that points to subdirectory bucket s.

```
Algorithm 23

DIRECTORY_REGION(cell,content,smallest,rcell)
{
   for(i = 0; i <= smallest.level; i++){
      Calculate the region that encloses 'smallest' at
         level 'smallest.level - i' and store its
         information in rcell;
      for(all directory entries){
         if(there is an entry which is equal rcell)
            return;
      }
   }
}
```

An insertion of a record may cause the databucket to overflow. In the BANG file structure, the splitting of a bucket is done until the best balance condition is reached. The splitting strategy for a subdirectory bucket is similar to the splitting of databucket. Suppose R is a region of the root directory that points to an overflow subdirectory bucket B. Let regions R1 and R2 be the results of the splitting of region R and the buckets B1 and B2 be pointed to by regions R1 and R2. All entries in bucket B that enclosed region R1 are moved into bucket B1 and the rest are moved to bucket B. If there is an entry E in a subdirectory bucket B with a region that is enclosed in region R1 and region R2 as shown in Figure 23.c, then the region of the entry E and the corresponding databucket should be split into two, the first is enclosed in region R1 and the other is enclosed in region R2.

Algorithm 24

```
SPLIT()
{
   Split the databucket;
   Write both buckets into the file;
   Find the unique <region,level> pair for both new and
                    old directory entries;
   If(subdirectory is not overflow) {
      Insert the new directory entry and modify old
           entry;
      Sort the directory entries;
      Write directory bucket;
   }
   else
      Split subdirectory bucket and modify root
                                          directory;
}
```

Algorithm 25

```
SPLIT_DIRECTORY_BUCKET()
{
   Find the splitting regions R1 and R2;
   if(there is a region E shared by R1 and R2)
      Split the region E and the corresponding
                                   databucket;
   Split the subdirectory bucket;
   Find the unique <region,level> for R1 and R2;
   Remove the entry R from and insert the new entries
                    R1 and R2 into the root directory;
   Sort the entries in the root directory;
}
```

The deletion of a record from a databucket or the deletion of a directory entry from a subdirectory bucket may cause the population of the bucket to fall below a certain threshold. In order to maintain the occupancy rates of the bucket, a merging attempt is done to merge the underflowing bucket with another bucket. The following algorithm shows how the merging is performed.

Algorithm 26

```
MERGING_DATA_BUCKET()
{
   do until no regions can be merged {
      Merge with a region that is immediately included
                                  in region to be merged;
      if(!success) {
         Merge with its buddy region ;
         if(!success)
            Merge with a region that is immediately
                     enclosed the region to be merged;
      }
   }
}
```

Given a region R at level L, the buddy of this region can be calculated by using the following Algorithm.

Algorithm 27

```
BUDDY()
{
   reg = 2 ** (L - 1);
   if(R > reg)
      buddy = R - reg;
   else
      buddy = R + reg;
}
```

In our implementation of the BANG file structure, we provide two types of query, the exact match query and the range query. Algorithm 28 shows how the range query is performed.

Algorithm 28

```
RANGE_RETRIEVAL(low,high,cond_l,cond_r)
{
   for(all root directory entries){
      if(the entry is included in query region){
         Read subdirectory bucket;
         for(all entries in the subdirectory bucket){
            if(the entry is included in query region)
               Read databucket and retrieve all records
                                        that match the query;
         }
      }
   }
}
```

Performance evaluation

It this section, we discuss the performances of the grid file and the BANG file structures. Based on the implementation methods of both file structures as discussed in Chapters III and IV, we expect that the BANG file has some advantages over the grid file.

1. In the BANG file structure, the expansion rates of the directory elements is proportional to the expansion rates of the disk blocks. Each directory entry corresponds to a disk block. In the grid file structure, the splitting of a disk block increases the number of directory elements rapidly. Figure 28

shows the example of the growth of the grid file during splitting. Before splitting the directory contains 24 elements(Figure 28.a). If the databucket corresponding to the starred directory element overflows, the bucket is split, and the directory elements are increased to 30(Figure 28.b).

2. During the splitting, the contents of two newly-formed buckets are almost balanced for the BANG file structure. Therefore, we expect better storage utilization, especially for a nonuniform data distribution.

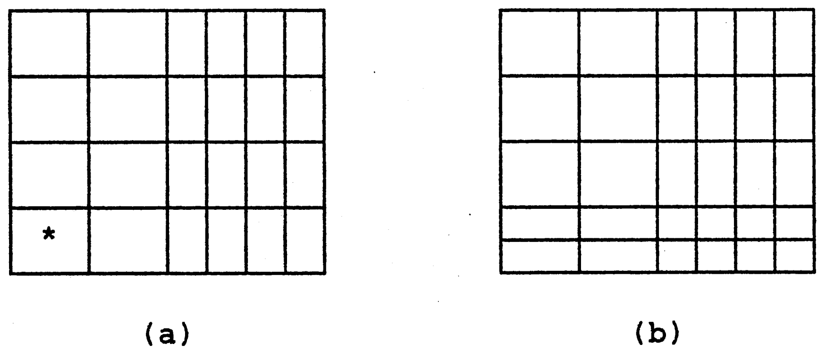


Figure 28. The Growth of the Directory.

In this thesis, we use the discrete event simulation method to predict the behavior of the grid file and the BANG file structures. To compare the performances of these two file structures, two programs are developed to simulate the behavior of these file structures in response to different sets of data. The specifications of our simulation studies are as follows.

1. Two simulation programs for grid file and BANG file structures were implemented in C language on the IBM PC/AT.
2. The database files were used to compare the performances of the two file structures consist of records with 3 attributes with the key range of $0 <= x < 16384$. The values of each attribute follow a poisson, normal or uniform distribution.
3. The data and subdirectory buckets are stored in disk blocks. The size of each block is fixed. A disk block is split if it overflows. If the contents of a disk block falls below a certain threshold, a procedure is invoked in order to merge two blocks to maintain reasonable storage utilization. The number of disk blocks needed to hold all records determines the storage performance of the file structure.
4. The statistics of the databucket utilization is recorded for every 200 insertions. The rate of the directory entry overhead is only recorded for the grid file structure. Each bucket in the BANG file structure is pointed to by exactly one directory entry. Therefore, there is no directory entry overhead for BANG file structure.

The following table shows some information about the grid and BANG file system as used in our experiment.

Dimension	4
Length of data and directory bucket	2048 bytes
Records per databucket	64
Number of inserted records	10000 - 16000

To compare the performances of the grid file and the BANG file structures, we examine both file structures applied to four databases with types of data distribution. Each database contains 3 dimensional records with the distribution as shown in Table II. The databucket utilization and the number of directory entries per databucket for the grid file and the BANG file structures in response to the various test files are shown in Table III.

TABLE II

DATA DISTRIBUTION

Test file	Dimension			Number of records
	1	2	3	
1	Fig. 29	similar	similar	10000
2	Fig. 32	similar	similar	10000
3	Fig. 35	similar	similar	16000
4	Fig. 38.a	Fig. 38.b	Fig. 38.c	10000

TABLE III

DATABUCKET UTILIZATION

Test file	Average databucket utilization		Average # of subdirectory entries per databucket	
	grid	BANG	grid	BANG
1	69.6%	69.5%	1.14	1
2	60.0%	67.6%	1.86	1
3	51.4%	68.3%	2.18	1
4	58.7%	68.5%	1.99	1

Based on the observation of Figures 29 to 40 and the table above, we find that the BANG file structure has a consistent performance in databucket utilization for all types of data distributions. This is not surprising since the BANG file structure always divides the overflowing databucket into two balanced databuckets. The databucket utilization of the grid file structure depends on the type of data distribution. In uniform data distribution as shown in Figure 29, both grid and BANG file structures have the same performances in bucket utilization. In the database with the attributes that are clustered in a small range of data space as shown in Figure 33, 36 and 39, the BANG file structure has a better bucket utilization over the grid file structure. Table III also shows the average number of subdirectory entries per databucket. For all types of data distributions the BANG file structure maintains one directory entry per databucket. The number of directory entries per databucket for the grid file depends on type of data distribution. If the distribution of data is uniform then grid file has approximately one directory entry per databucket. For a non uniformly distributed database, the grid file has approximately two directory entries per databucket.

Tables IV to VII show the statistics for the range queries which are uniformly distributed within the range of the attribute. For each of the given range sizes, the results are averaged over 100 randomly generated queries. The range denotes the ratio of the size of the range specified in the query to the size of the range of all values of the attribute. The results of our experiments show that the performances of the grid file and the BANG file structures depend on the type of data distribution. For test file 1, which is uniformly distributed, both file structures have the same query performance. For test file 3, which is normally distributed, the performance of the BANG file is superior over the grid file. For test file 2 and test file 4, the performances of both file structures depend on the range size of the query. The grid file has a slight superiority over the BANG file in the small range size, and vice versa for the large range size.

TABLE IV

RANGE RETRIEVAL OF TEST FILE I

Range size	5%	10%	20%	25%	30%
Records found	1.41	10.39	80.12	155.52	267.9
Buckets accessed for grid file	3.58	5.70	14.42	20.82	28.07
Buckets accessed for BANG file	3.60	5.73	14.54	20.96	28.10

TABLE V

RANGE RETRIEVAL OF TEST FILE II

Range size	5%	10%	20%	25%	30%
Records found	1.07	8.20	54.37	85.49	118.12
Buckets accessed for grid file	2.75	3.79	7.12	8.62	11.57
Buckets accessed for BANG file	3.29	3.98	6.15	7.22	8.87

TABLE VI

RANGE RETRIEVAL OF TEST FILE III

Range size	5%	10%	20%	25%	30%
Records found	2.68	25.77	239.98	516.96	1636.24
Buckets accessed for grid file	3.71	6.48	22.09	38.34	84.11
Buckets accessed for BANG file	3.52	5.40	16.22	28.24	68.88

TABLE VII

RANGE RETRIEVAL OF TEST FILE IV

Range size	5%	10%	20%	25%	30%
Records found	1.47	11.67	73.21	122.86	176.24
Buckets accessed for grid file	2.95	4.22	9.35	12.99	17.65
Buckets accessed for BANG file	3.50	4.51	8.49	11.09	14.33

Based on the observation on the behaviour of both file structures in response to the range query, we find that in the clustered area of the records, the bucket utilization of the grid file is better that the BANG file. The query performance of the grid file structure is better than the BANG file structure for range queries with small range sizes which are concentrated in the clustered area of the data space. If the range of the queries is large and the queries are uniformly distributed over the whole area, then the query perfomance of the BANG file structure is better than the grid file structure.

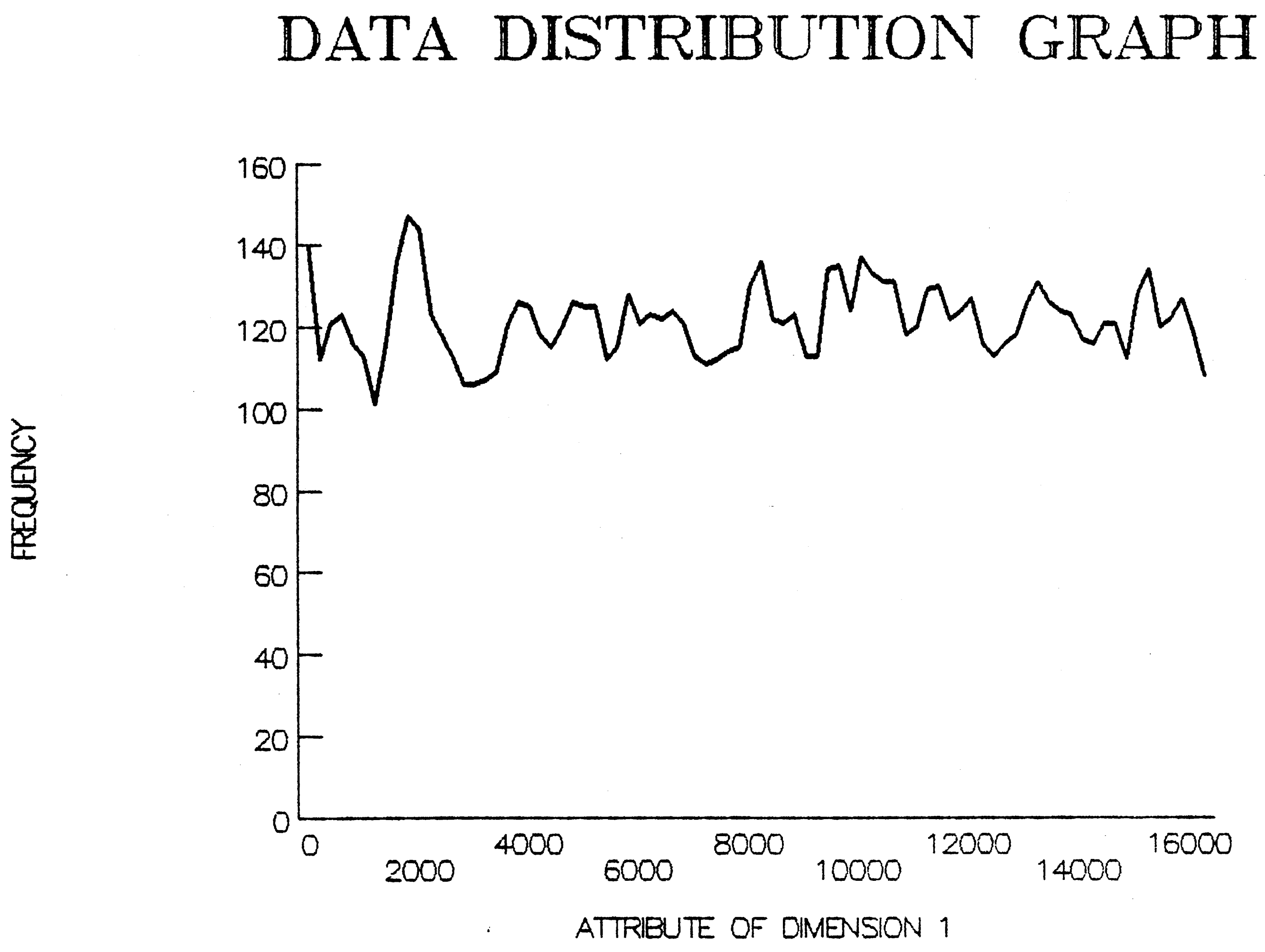


Figure 29. Data Distribution of Test File 1.
(Similar for all Dimension)

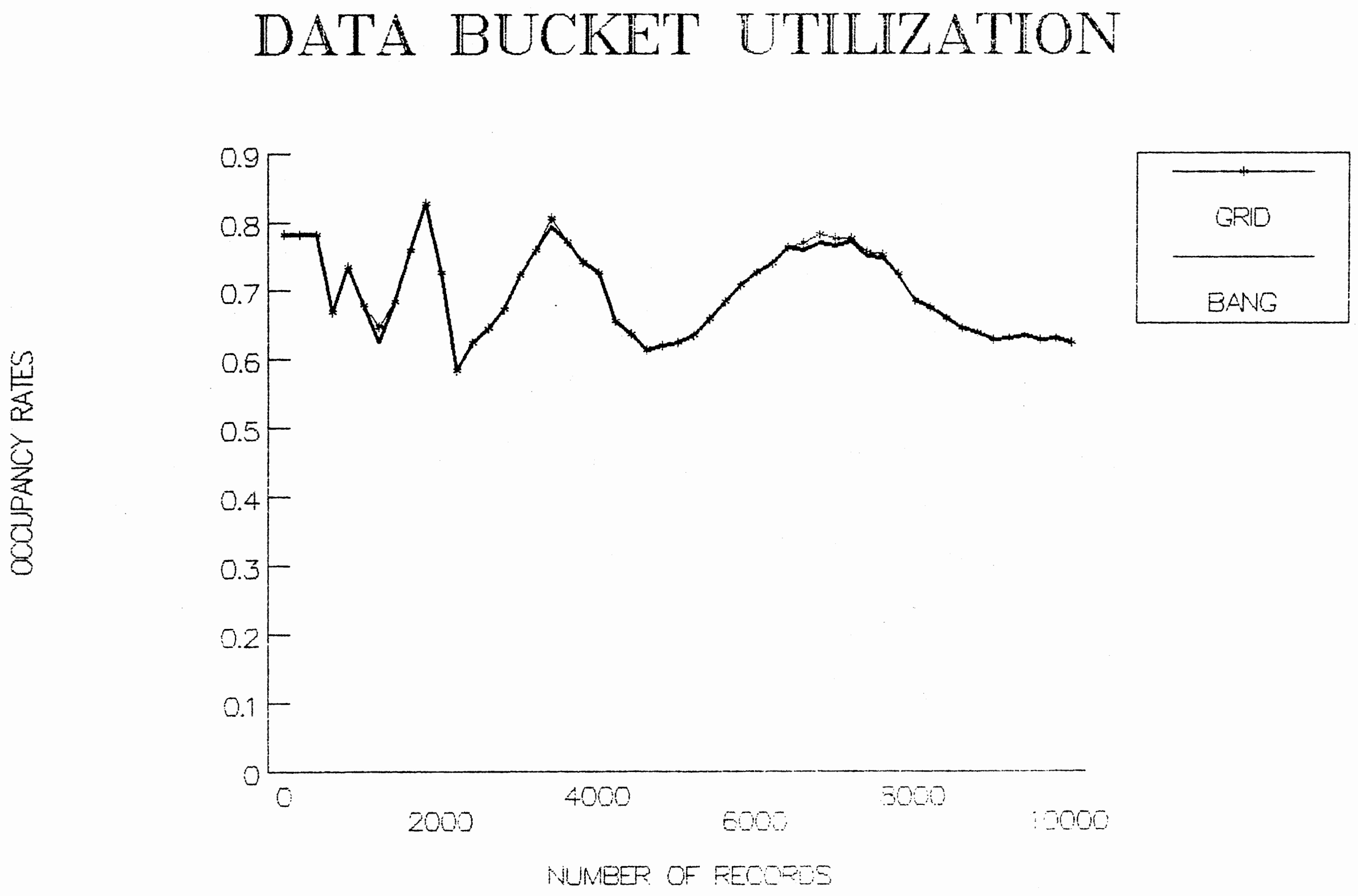


Figure 30. Databucket Utilization for Test File 1.

SUBDIRECTORY POINTER OVERHEAD

FOR GRID FILE

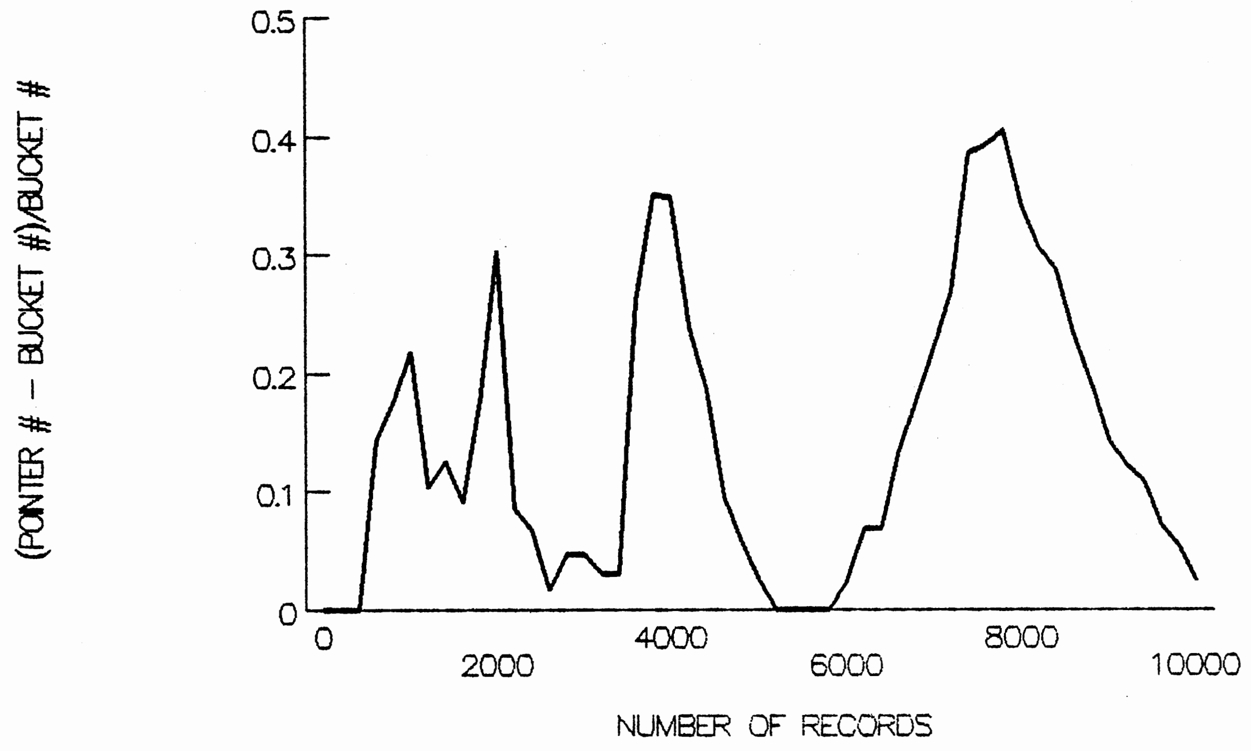


Figure 31. Pointer Overhead for Test File 1.

DATA DISTRIBUTION GRAPH

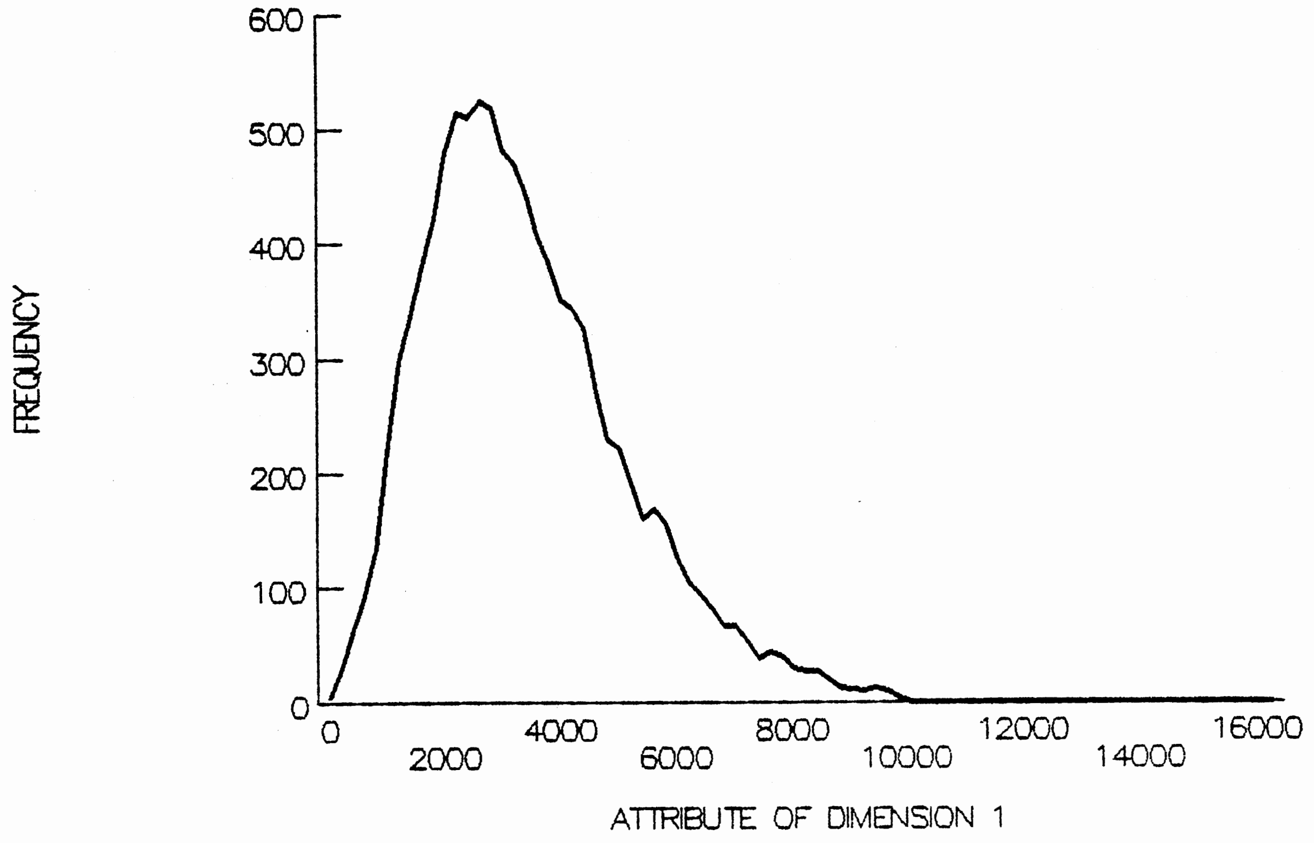


Figure 32. Data Distribution of Test File 2.
(Similar for all Dimension)

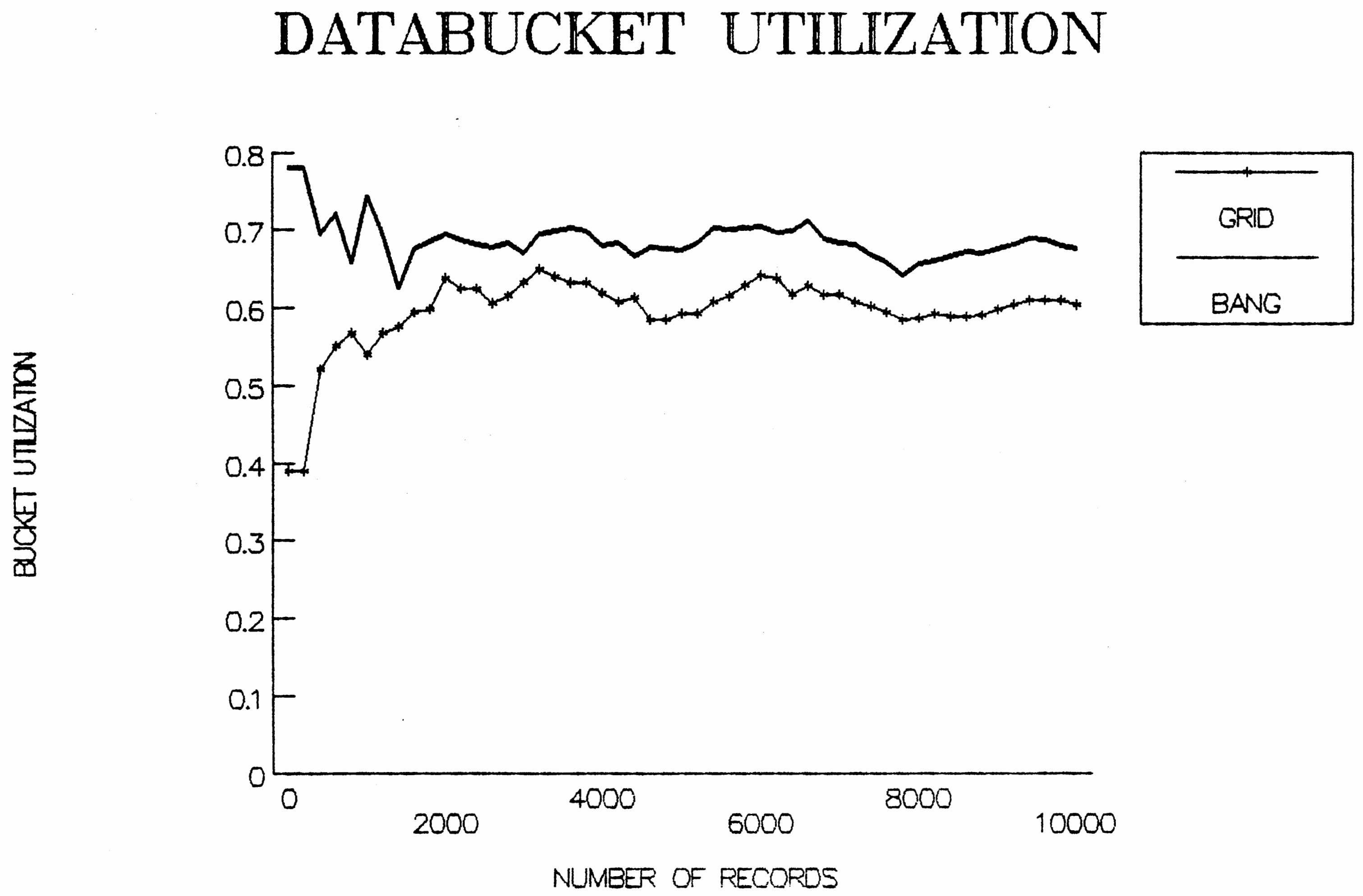


Figure 33. Databucket Utilization for Test File 2.

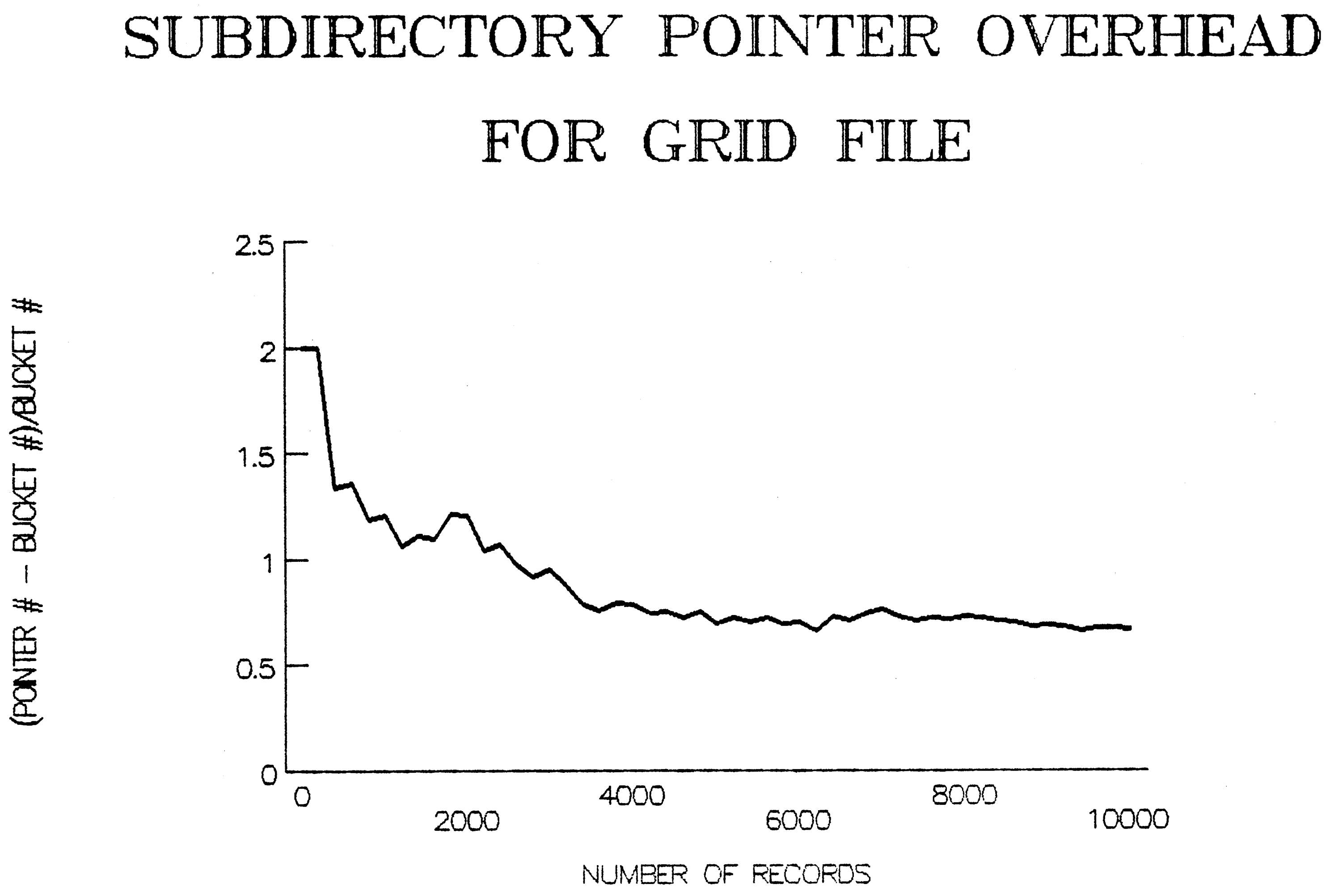


Figure 34. Pointer Overhead for Test File 2.

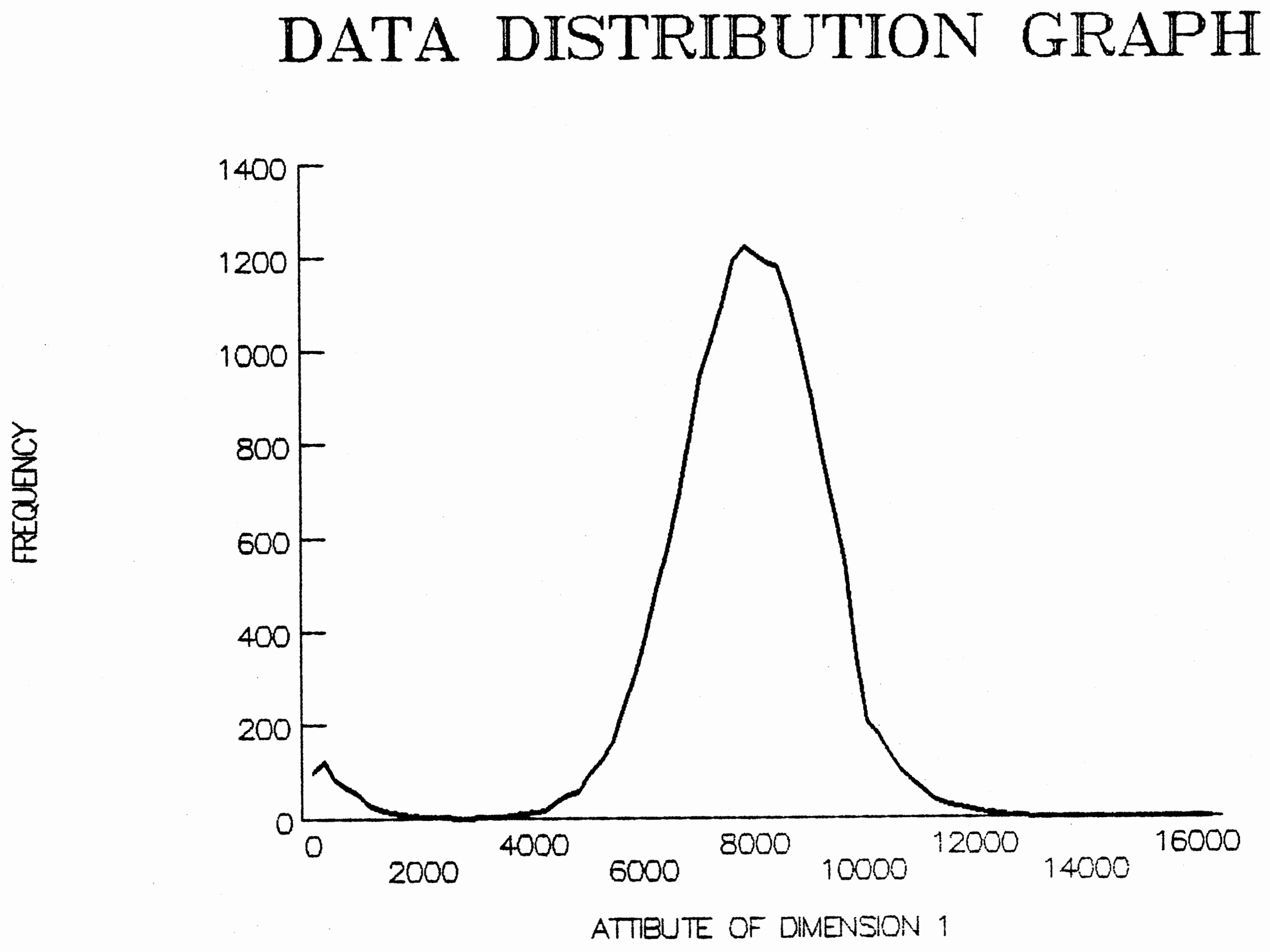


Figure 35. Data Distribution of Test File 3.
(Similar for all Dimension)

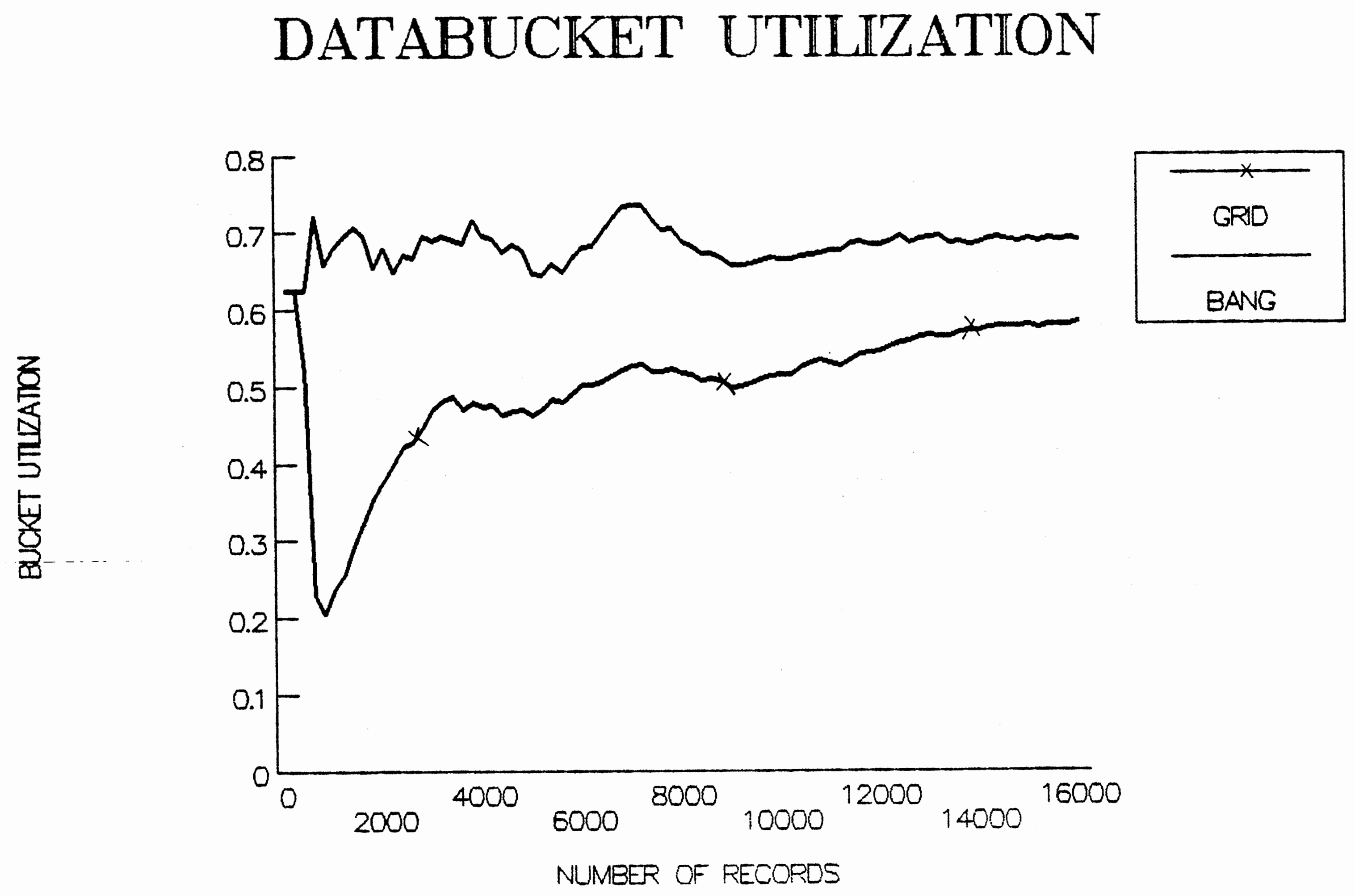


Figure 36. Databucket Utilization for Test File 3.

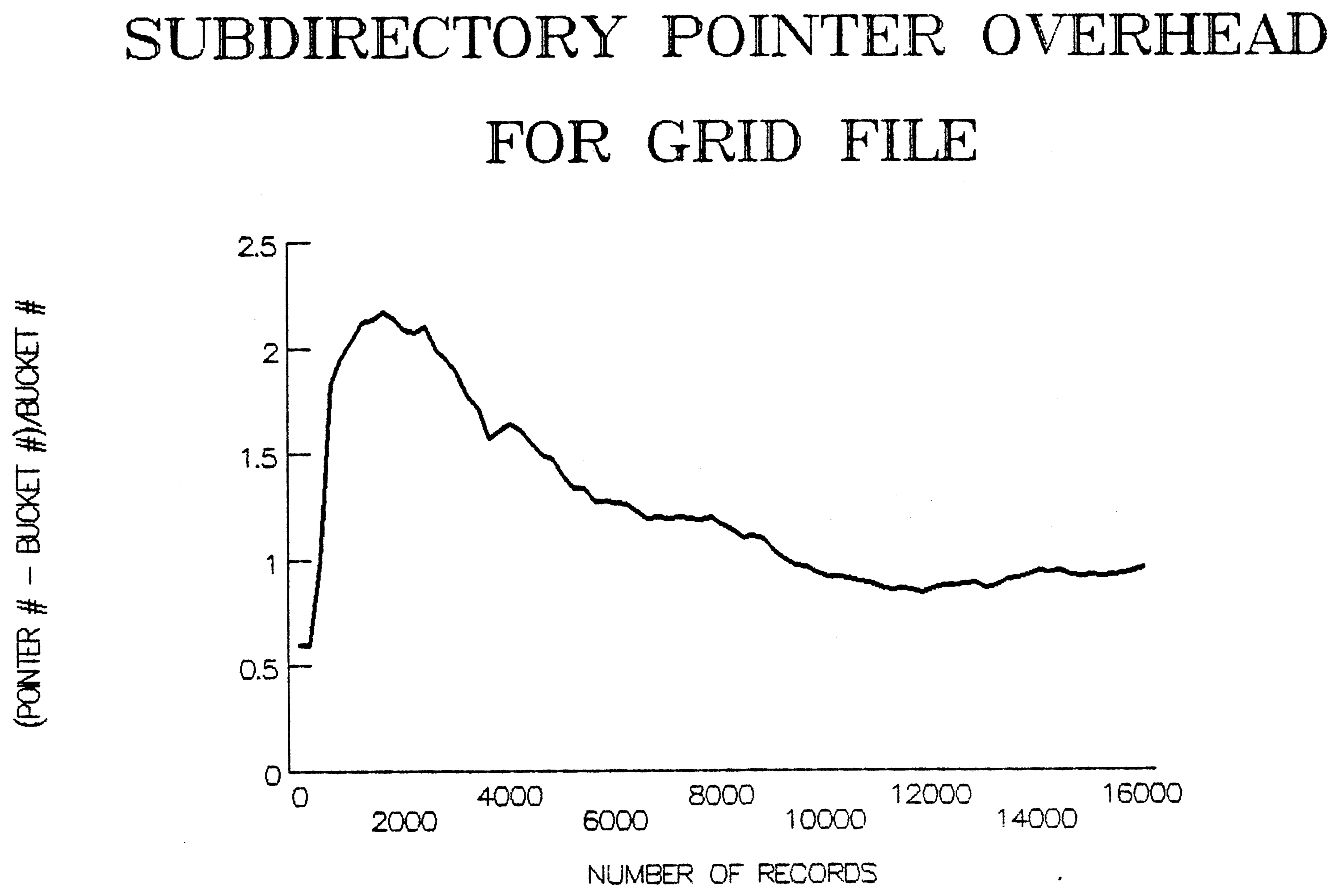
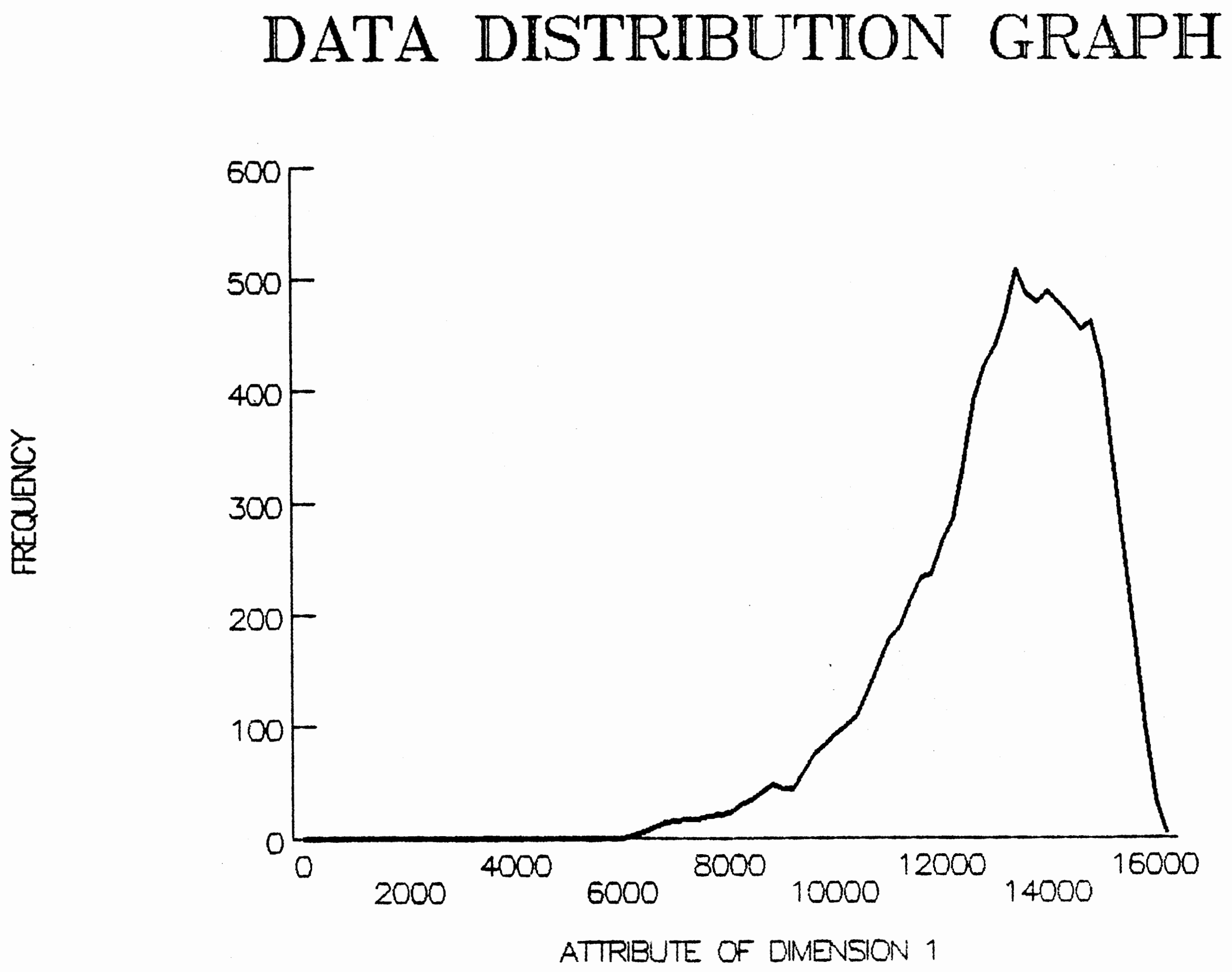
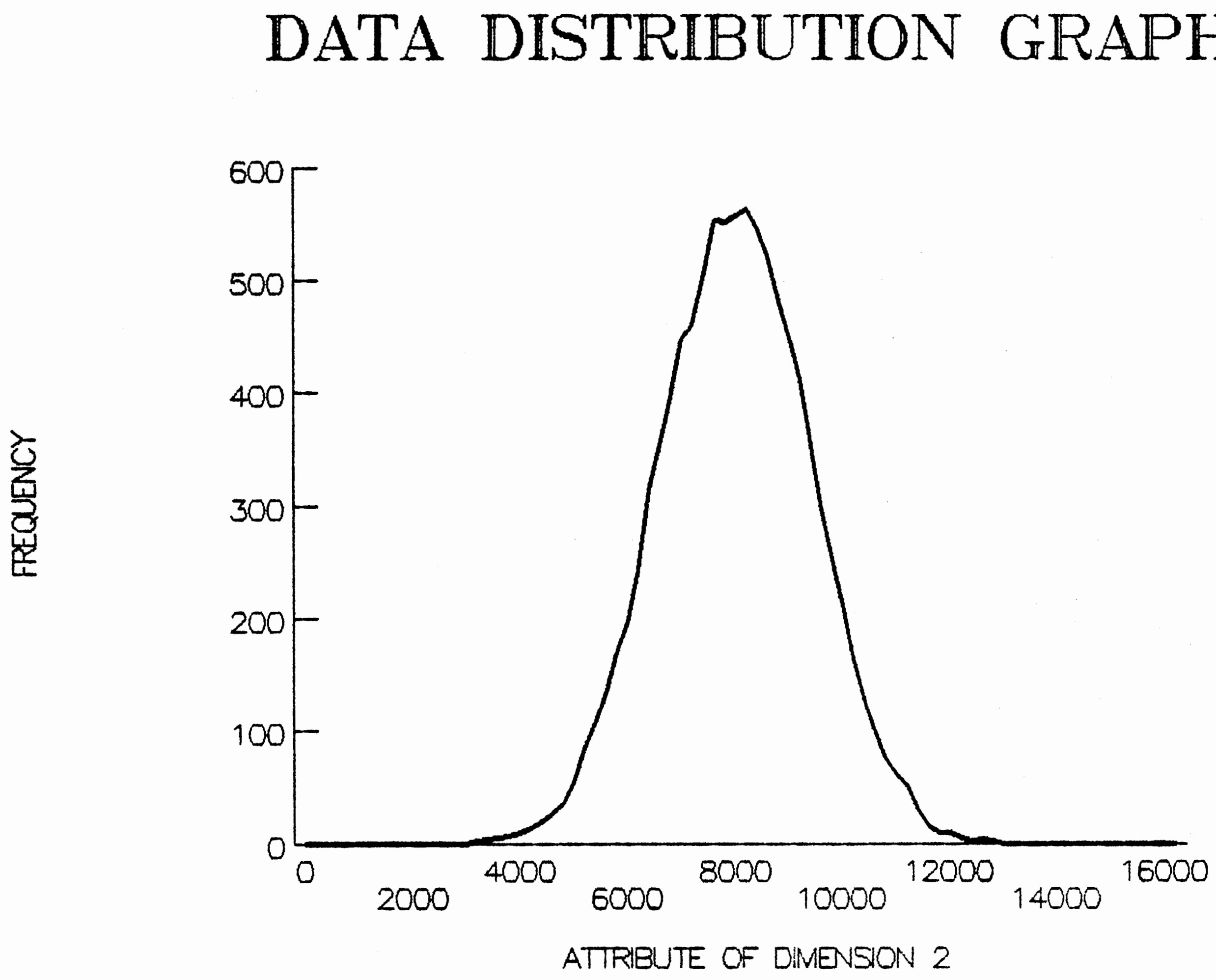


Figure 37. Pointer Overhead for Test File 3.



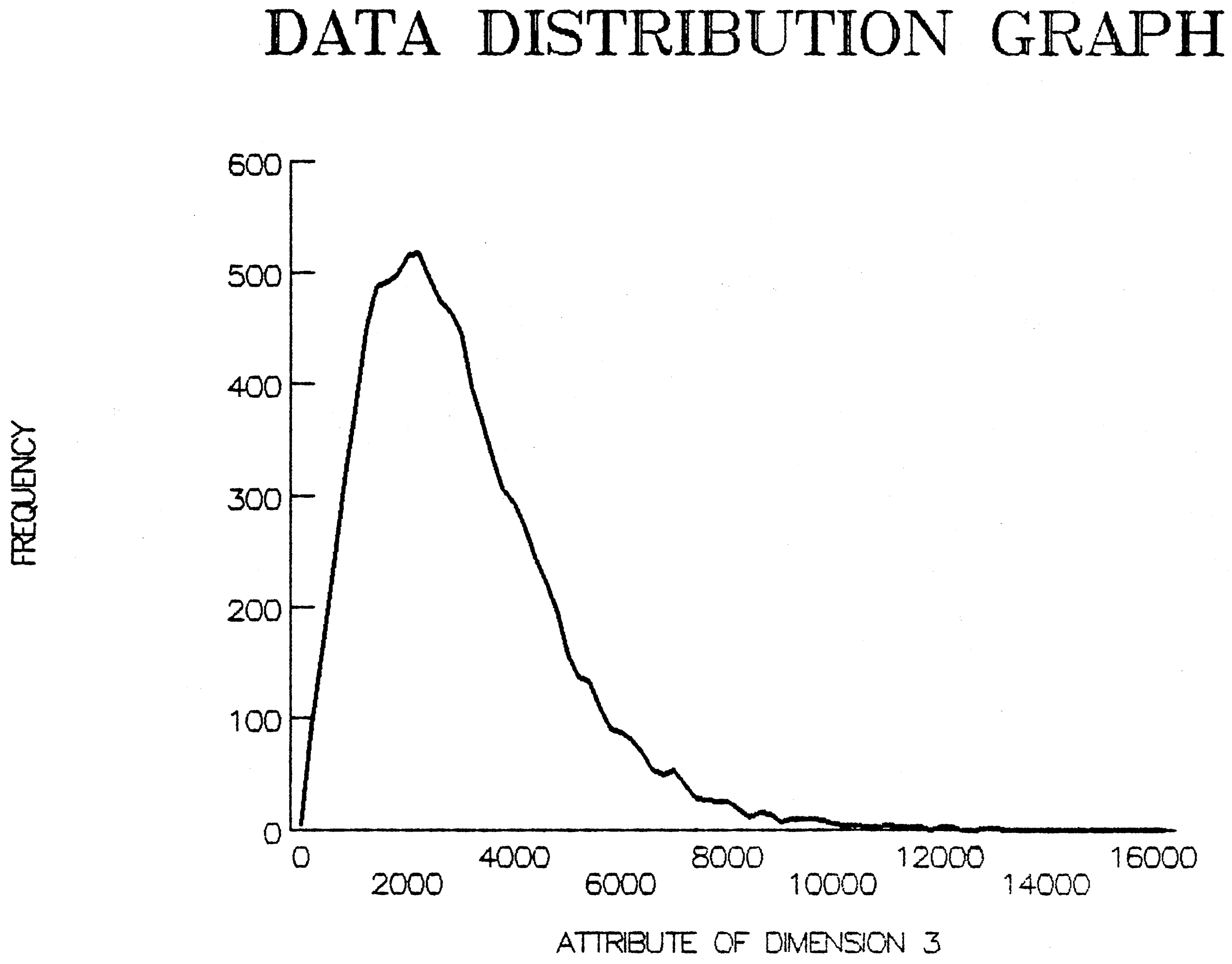
(a) Dimension 1.

Figure 38. Data Distribution of Test File 4.



(b) Dimension 2.

Figure 38 (cont)



(c) Dimension 3.

Figure 38 (cont)

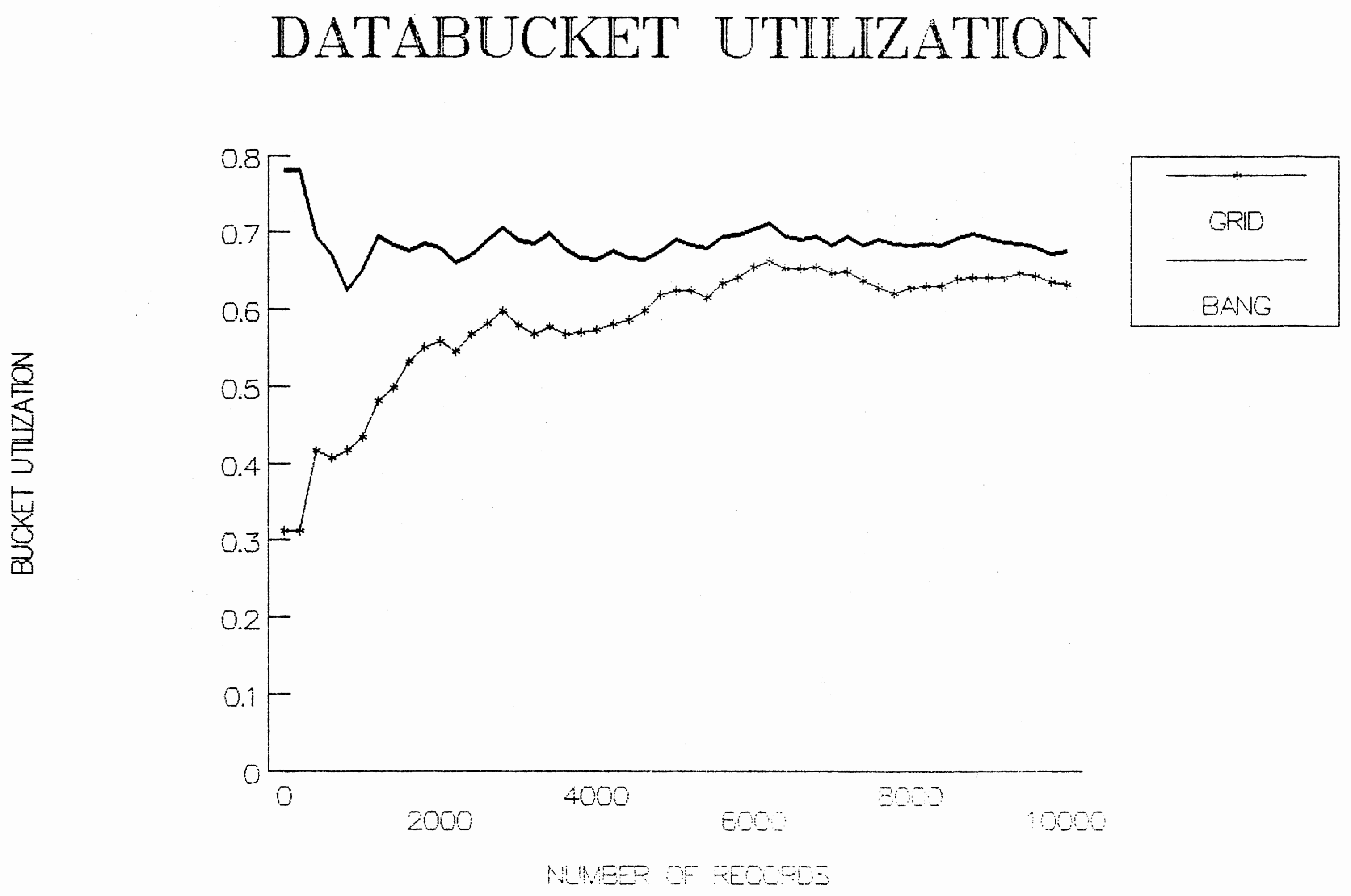


Figure 39. Databucket Utilization for Test File 4.

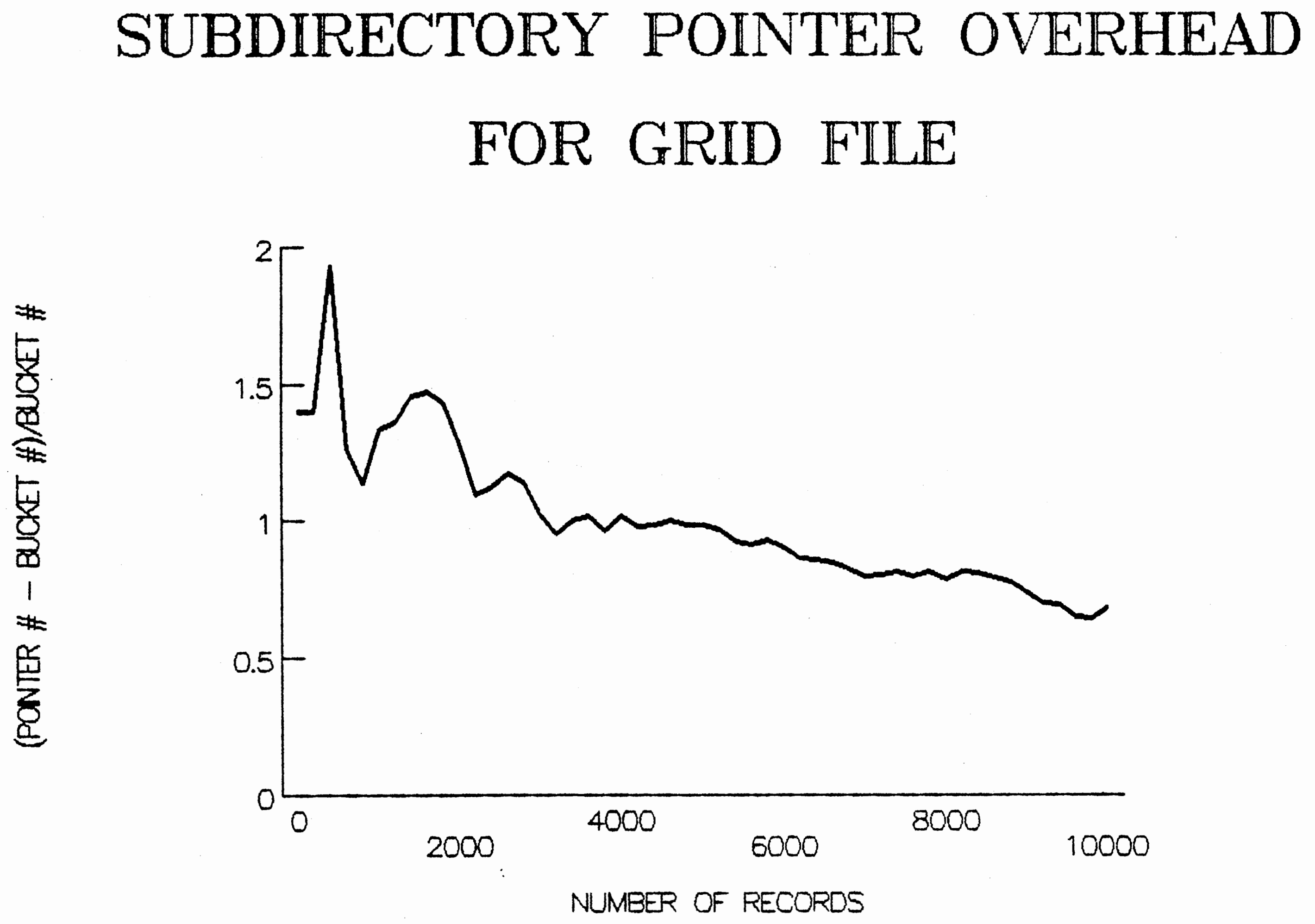


Figure 40. Pointer Overhead for Test File 4.

CHAPTER VI

SUMMARY AND CONCLUSIONS

In this thesis we compare the performances of two types of index-based file structures, the grid file and the BANG file structures, which are suitable to organize large, dynamic, k-dimensional records. Both file structures have a good adaptation to the change of data distribution. An overflow is handled by splitting the bucket into two and an underflow is handled by a merging operation. The main difference between the two file structures is the method of handling an overflow databucket. In the grid file structure, an overflow databucket is split into two. The splitting operation is completed as soon as the overflow condition is removed. In the BANG file structure, an overflow databucket is split into two balanced databuckets.

Conclusions of our simulation studies are as follows.

1. The databucket utilization and the subdirectory entry overhead of the grid file structure is influenced by the type of data distribution. This influence is not so obvious in the BANG file structure.

2. In the uniform distributed database, both file structures have the same performance in databucket utilization and the number of buckets accessed during the range query.
3. The more the distribution of records in the database differs from uniform, the better the BANG file in databucket utilization and subdirectory entry overhead.
4. In non uniform database, the grid file structure has a better performance than the BANG file structure for region queries with small range sizes which are concentrated in the area where the records are clustered. If the range size is large and the query region is uniformly distributed over the data space, then the BANG file has a better performance than a the grid file structure.

BIBLIOGRAPHY

1. Bentley, J. L. "Multidimensional Binary Search Trees Used for Associative Searching." Communications of the ACM 18, 9(1975), 509-517.

2. Bentley, J. L. "Multidimensional Binary Search Trees in Database Applications." IEEE Trans. on Software Engineering SE-5, 4(July 1979), 333-340.

3. Burkhard, W. A. "Interpolation-Based Index Maintenance." In Proc. ACM Symp. Principles of Database Systems (1983), 76-85

4. Casey, R. G. "Design of Tree Structures for Efficient Querying." Communications of the ACM 16, 9(1973), 549-556.

5. Claybrook, B.G., Claybrook, A.M., Williams, J. "Defining Database Views as Data Abstractions." IEEE Trans. on Software Engineering SE-11, 1 (Jan 1985), 3-14.

6. Finkel, R. A., Bentley, J. L. "Quad Trees, a data structure for Retrieval on Composite keys." Acta Informatica 4, 1(1974), 1-9.

7. Freeston, M. "The BANG File: A New Kind of Grid File." Proc. ACM SIGMOD 1987 Annual Conference SIGMOD Record 16, 3(Dec. 1987), 260-269.

8. Han, C. C. "A Grid File Approach to Large Multidimensinal Dynamic Data Structure." Unpub MS Thesis, Oklahoma State University, (Dec 1987).

9. Hinrichs, K. "Implementation of The Grid File Design Concepts and Experience." BIT 25, 3(1985), 569-582.

10. Knuth, D. E. "The Art of Computer Programming Vol. 3: Sorting and Searching." Addison-wesley, Reading, Mass. 1973, 550-567.

11. Kriegel, H. P. "Performance Comparison of Index Structures for Multi-key Retrieval." Proc. ACM SIGMOD, BOSTON, Massachusetts (1983), 186-196.

12. Litwin, W. "Linear Hashing : A New Tool for File and Table Addressing." Proceedings Sixth International Conference on Very Large Databases, Montreal, Canada (1980), 212-223.

13. Lum, V. Y. "Multi-attribute Retrieval with Combined Indexes." Communications of the ACM 13, 11(1970), 660-665.

14. Nievergelt, J., Hinterberger, H., Sevcik, K.C. "The Grid File: An Adaptable, Symmetric Multikey File Structure." ACM Trans. on Database Systems 9, 1 (1984), 38-71.

15. Orenstein, J. A. "Multidimensional Tries Used for Associative searching.", Information Processing Letters 14, 4(June 1982), 150-157.

16. Otoo, E. J. "A Multidimensional Digital Hashing Scheme for Files With Composite Keys." Sigmod Rec. (USA), 14, 4(Dec 1985), 214-229.

17. Otoo, E. J., Merrett, T. H. "A Storage Scheme for Extendible Arrays." Computing 31, 1(1983), 1-9.

18. Ouksel, M., Scheuermann, P. "Multidimensional B-Trees: Analysis of Dynamic Behavior.", Bit 21(1981), 401-418.

19. Ouksel, M., Scheuermann, P. "Storage Mappings for Multi-dimensional Linear Dynamic Hashing.", In Proc. ACM Symp. Principles of Database Systems(1983), 90-105.

20. Peterson, J. L., Norman, T. A., "Buddy Systems." Communications of the ACM 20, 6(1977), 421-431.

21. Regnier, M. "Analysis of Grid File Algorithms." BIT 25(1985), 335-357.

22. Robinson, J. T. "The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes." Proc. ACM SIGMOD, Ann Arbor, Michigan (1981), 10-18.

23. Saritepe, H.N.A. "An Analytical Comparison of Grid File and K-D-B-Tree Structures." Unpub. MS Thesis, OSU (Dec. 1987).

24. Scheuermann, P., Ouksel, M. "Multidimensional B-Trees for Associative Searching in Database Systems." Information system 7, 2(1982), 127-137.

25. Vallarino, O., "On The Use of Bit Maps for Multiple Key Retrieval." ACM SIGPLAN Notices 11, (1976 Special Issue), 108-114.

VITA

Tiong-Hu Lian

Candidate for the Degree of

Master of Science

Thesis: IMPLEMENTATION AND EVALUATION OF BALANCED AND NESTED GRID (BANG) FILE STRUCTURES

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Padang, Indonesia, March 22, 1960, the son of Mr. and Mrs. Tjin-Siong Lian.

Education: Graduated from Indonesia High School, in April 1979; received Bachelor of Science in Civil Engineering from National Taiwan University in 1983; completed requirements for the Master of Science degree in Computing and Information Science at Oklahoma State University in December, 1988.

Professional Experience: Field supervisor at R.S.E.A. construction company, Indonesia, 1984-1985; Graduate assistant at Oklahoma State University, Computing and Information Science Department, January 1988 to May 1988.