A GRID FILE APPROACH TO LARGE MULTIDIMENSIONAL DYNAMIC DATA STRUCTURES

BY

CHANG CHUN HAN

Bachelor of Science

Seoul National University

Seoul, Korea

1975

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
May, 1988

A GRID FILE APPROACH TO LARGE MULTIDIMENSIONAL DYNAMIC DATA STRUCTURES

Thesis Approved:

Donald D Fisher
Thesis Adviser

J P Chandler

Thomas C

Norman N. Durham
Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my sincere respect and appreciation to my major advisor, Dr. Donald D. Fisher, for his guidance on this study and also for his warm encouragement during my academic career in Oklahoma State University. I would like to extend my thanks to Dr. John P. Chandler and Dr. K. M. George for serving as members of my graduate committee.

I thank my father, Young Poong Han, for his perpetual confidence in me. I am especially indebted to my beloved children, You Jin and Eun Sung, and to my wife, Sang Sook, for their love and understanding.

I am grateful to the Korea Long Term Credit Bank, Seoul, Korea, which provided me with a chance to study abroad with financial support.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

CHAPTER I

INTRODUCTION

In the data processing environment, data can be represented in a variety of ways. The structure ultimately chosen for a specific task for representing the data is heavily influenced by the type of operations to be performed on the data set and by its volume. There has been much progress in designing graceful data structures. Major advances may be found in such structures as balanced trees and dynamic forms of hashing[16]. However, there have been increasing demands to develop efficient structures to meet the diversified requirements of modern information society. Commomplace yet complicted queries such as "Find all records associated with black women aged from 20 to 30 who have an annual income below $15,000 and live in the 10 southern states." require thoughtful design of the underlying data and file structures. Although it seems to be difficult to get a single solution for complex information demands, there appeared recently several attempts to design efficient data or file structures for that kind of problem.

We call the structures "large multidimensional dynamic structures". By 'large', we mean that the data set stored is so large that the bulk of data must remain on secondary

storage, usually random access devices, even when operations are being done. By 'multidimensional', we mean that a given data set F consists of records R each of which is a k-dimensional key vector such that R = (a_1, a_2,..., a_k), where k is a positive integer. By 'dynamic', we mean that we can execute common operations on the data set such as FIND, INSERT, DELETE, UPDATE and some kinds of range search on-line.

Recently the grid file[27] was presented as an attempt to support efficient operations on multikey processing fields. It has many interesting properties which might be useful to cover the deficiencies of conventional inverted files. An inverted file is a popular structure in the practical application of multikey processing. But it has severe space and time overhead to maintain sorted index lists for each key type and to perform boolean operations.

The grid file is one of the grid-cell-type data structures that organizes the data space in which a set of given objects is embedded. It is as a whole a symmetric, adaptable file structure designed to process large amounts of multidimensional data efficiently[27]. It is symmetric in the sense that every key field is treated with the same efficiency ,i.e., the grid file does not have a primary key representing a record distinctly. 'Adaptable' means that the data structure adapts its shape automatically to the content it must store . The grid file adjusts its structure dynamically so that bucket occupancy and access time are uniform over the entire file, even though the data may be distributed highly nonuniformly

over the data space. With those properties, the grid file can perform efficient queries such as range and nearest neighbor queries on multidimensional data.

Nievergelt[26] groups organizing techniques into two broad classes: (1) those that organize the set of given objects to be processed and (2) those that organize the space in which those data are embedded. The difference can be seen between comparative search in binary search tree and radix search in a trie. The former is based on tree structure and the latter relies on address computation like a hashing method. In Chapter II, we review several structures in each category in view of multikey processing. We will cover quadtree[14], k-d tree[3,4], k-d-B tree[34] and multidimensional B-tree[38] which can be classified as class (1), and an extendible cell method[39], interpolated-index maintenance[6] for class (2). In Chapter III, we will describe the grid file [27] in general and define three abstract data types underlying grid file. In Chapter IV, we will esta- blish the major concepts and strategies for implementation of grid file structure. Chapter V will cover algorithms for the implementation of grid files. In Chapter VI, we will describe the structure of a program. We discuss the scope of this program such as available dimensions and key types, user functions, and available data set size. In Chapter VII, we will discuss concurrency problem. Chapter VIII will conclude this thesis with a summary of our overall design strategies and the results of simulation studies.

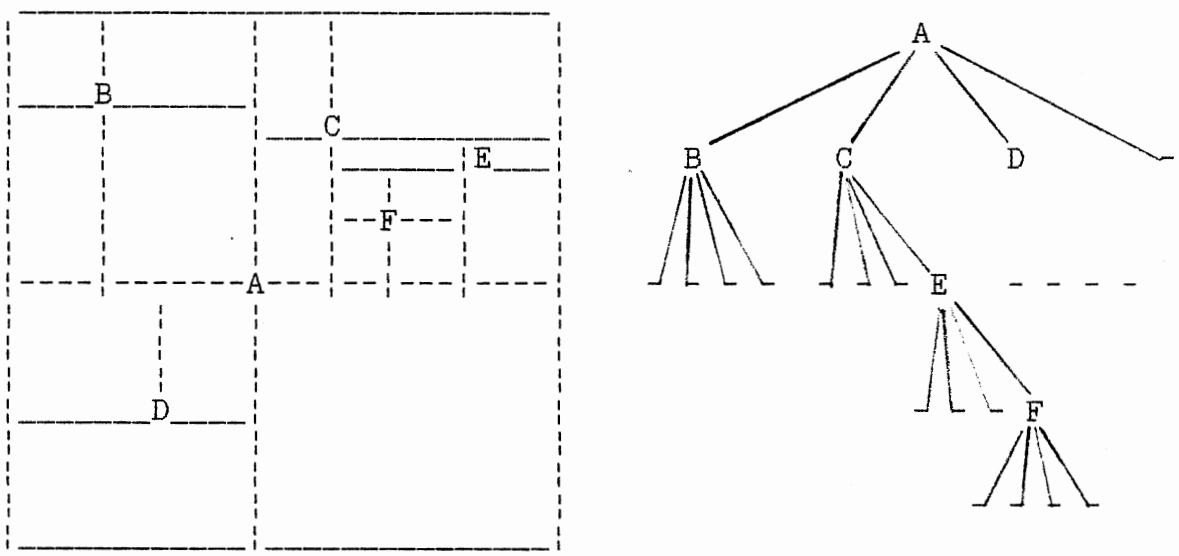
CHAPTER II

PREVIEW OF MULTIKEY PROCESSING TECHNIQUES

Tree-structured technique

(1) Point quadtree[14] is a two-dimensional generalization of a binary search tree. Each data point is a node in a tree having four sons, which are roots of subtrees corresponding to ordered quadrants(Figure-1). The process of inserting into a point quadtree is analogous to the scheme used for binary search trees. At each node of the tree a four-way comparison operation is performed and the appropriate subtree is chosen for the next test. Records are inserted at leaf nodes like in binary search tree. The tree may be unbalanced. Balancing the tree is quite complex; furthermore deletion of a node is more complex[36] than insertion. The problem with a large number of dimensions in a quadtree is that the branching factors becomes very large(2^k for k dimensions), thereby requiring much storage for each node as well as many null pointers for terminal nodes. While a quadtree has pointer overhead problem, it has an advantage that the comparison operation can be performed in parallel for the k values.

(2) A k-d tree[4] is a multidimensional version of the binary search tree with the distinction that at each level of the tree

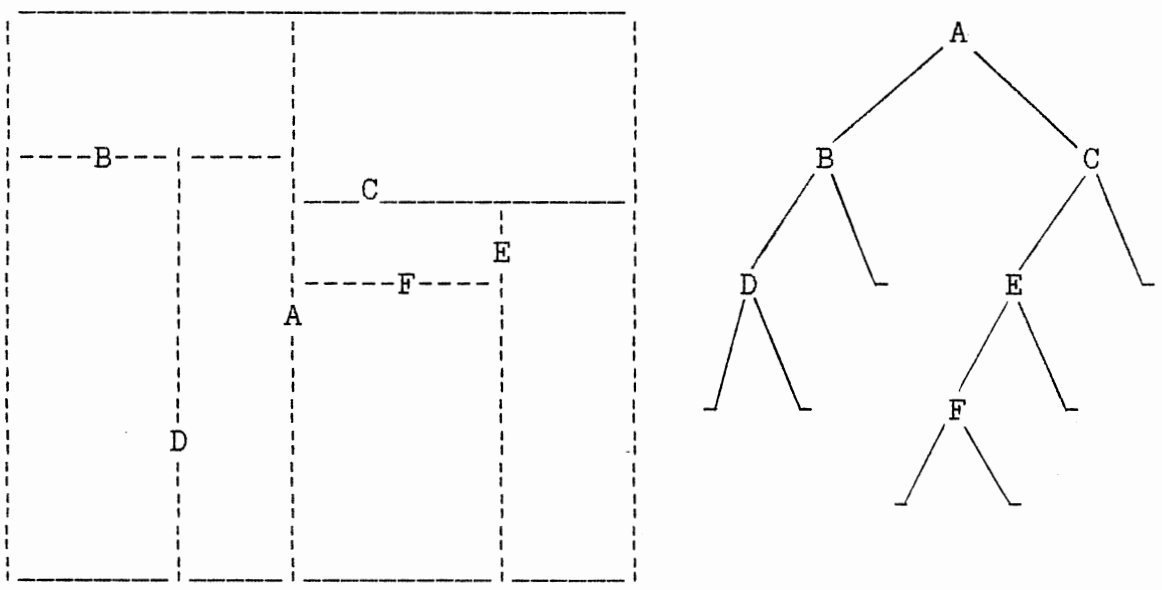


Record space Quad tree representation

Figure-1. Record space and its quad tree representation

different coordinate is tested for deciding the direction in which a branch is to be made. Two kinds of implementations of k-d trees are possible: homogeneous and nonhomogeneous k-d trees. A homogeneous k-d tree is a binary tree in which each record contains k keys, information, right and left pointers and one of k discriminators. In nonhomogeneous k-d trees, all records are stored in external nodes or buckets. To insert a new record into a k-d tree, we do a top-down search to find the insertion position by comparing at each node visited corresponding keys of the discriminator. A cyclic method is generally used for choosing a discriminator among k attributes. A simple 2-d tree using cyclic method for choosing the

discriminator is shown in Figure-2. However, for many kinds of searches one might get better performance by choosing as discriminator a certain key whose values are particularly well spread or by choosing a key which is often specified in queries.



Record space 2-d tree representation

Figure-2. Record space and its 2-d tree representation

Formally, the invariant of a k-d tree is that for any node A of a j-discriminator, all nodes in the left subtree of A have j-discriminator values less than A's j-discriminator value, and likewise all nodes in the right subtree have greater j- discriminator values. In contrast to single key binary

search trees, it seems to be very difficult to maintain balanced k-d trees dynamically.

(3) k-d-B tree[34] is one of solutions for retrieving multikey records via range queries from a large dynamic balanced index. A k-d-B tree is a data structure combining properties of k-d trees and B-trees. Hence k-d-B trees are multiway trees with fixed sized nodes that are always totally balanced in the sense that number of nodes accessed on a path from the root node to a leaf node is the same for all leaf nodes. A k-d-B tree partitions the search space recursively into two subspaces based on comparison with some elements of k discriminators. Like B-trees, k-d-B trees consist of a collection of pages. However, there are two types of pages in a k-d-B tree. One is region pages which contain a collection of (region, pageID) pairs and the other is point pages which contain a collection of (point, location) pairs, where location points to a bucket in secondary storage. The point pages are the leaf nodes of the tree. A 2-d-B tree showing region and point pages is presented in Figure-3.

The invariants of a k-d-b tree are:

1. Considering each page as node and each page ID in a region page as node pointer, the resulting graph structure is a multiway tree with a root page. Furthermore, no region page contains a null pointer, and no region page is empty.
2. The path length, in pages, from the root page to a leaf

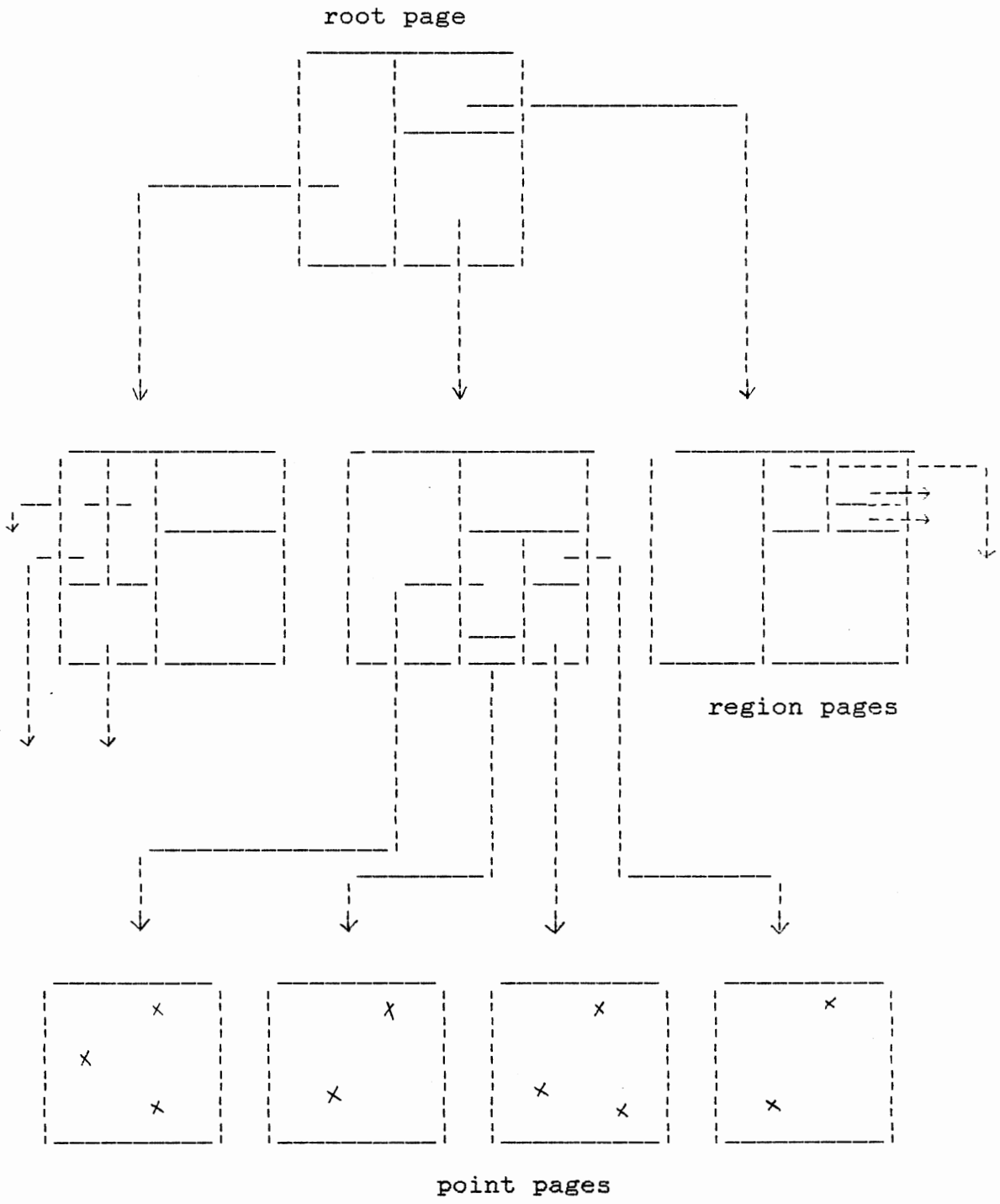


Figure-3. Region and point pages in a 2-d-B tree

page is the same for all leaf pages.

3. In every region page, the regions in the page are disjoint, and their union is a region.
4. If the root page is a region page(it may not exist, or if there is only one page in the tree it will be a point page), the union of its regions is $domain_0$ x $domain_1$ x ... $domain_{k-1}$.
5. If (region,child ID) occurs in a region page, and the child page referred to by child ID is a region page, then the union of the regions in the child page is a region.
6. If the child page is a point page, then all the points in the page must be in region.

To insert a new record into the tree, we search down the tree from root page to the point page(leaf node) and add (point,location) to the point page. If overflow occurs, we split the point page into two point pages and distribute the record's index appropriately into the pages. We should do back tracking like B-trees and split (region,pageID) in parent range page into two region index pointing two new index created one level below.

(4) Multidimensional B-tree[38] is an extension of the multiple-attribute-tree structure, in which the directory is a k-level tree, such that a unique path from the root to a leaf node corresponds to a distinct combination of the k-attributes.

However, this structure uses B-trees to maintain the filial sets at each level of the directory. A filial set at level i is the set of attribute A_i values appearing together with the same value of A_{i-1} in the whole data set. Each of the k-attributes of the data set to be indexed is represented by a separate level in this tree-directory and each node is of this tree is itself a B-tree. Root nodes of all filial sets at level i are linked together and an entry point, e.g. LEVEL(i), is provided to the beginning of each such linked list(Figure-4). There are two kinds of pointers in a node in the B-tree for an attribute A_i. One point to a node at the next level in the same B-tree containing values of A_i and the other points to a B-tree at level(i+1) which contains the sets of values of attribute A_{i+1}. There is an assumption that each node in a B-tree corresponds to a page on a secondary device. If the order of a B-tree is relatively small, the adjacent nodes in the tree can be grouped together on the same page in order to avoid severe low page occupancy.

The main concern in insertion is to maintain the ordering imposed on the filial sets at each level. If we are going to insert a record $R(a_1, a_2, \ldots, a_n)$ such that the combination $a_1, a_2, .., a_{i-1}$ already exists in the tree-directory, but $a_1, a_2, \ldots, a_i$ does not, we first insert a_i into the B-tree at level i determined by the combination $a_1, a_2, \ldots, a_{i-1}$. Then we insert a filial set, F_i, consisting of the single value a_{i+1} between the filial sets at level(i+1) associated with a_i'and a_i" assuming that the value of a_i is inserted between a_i' and

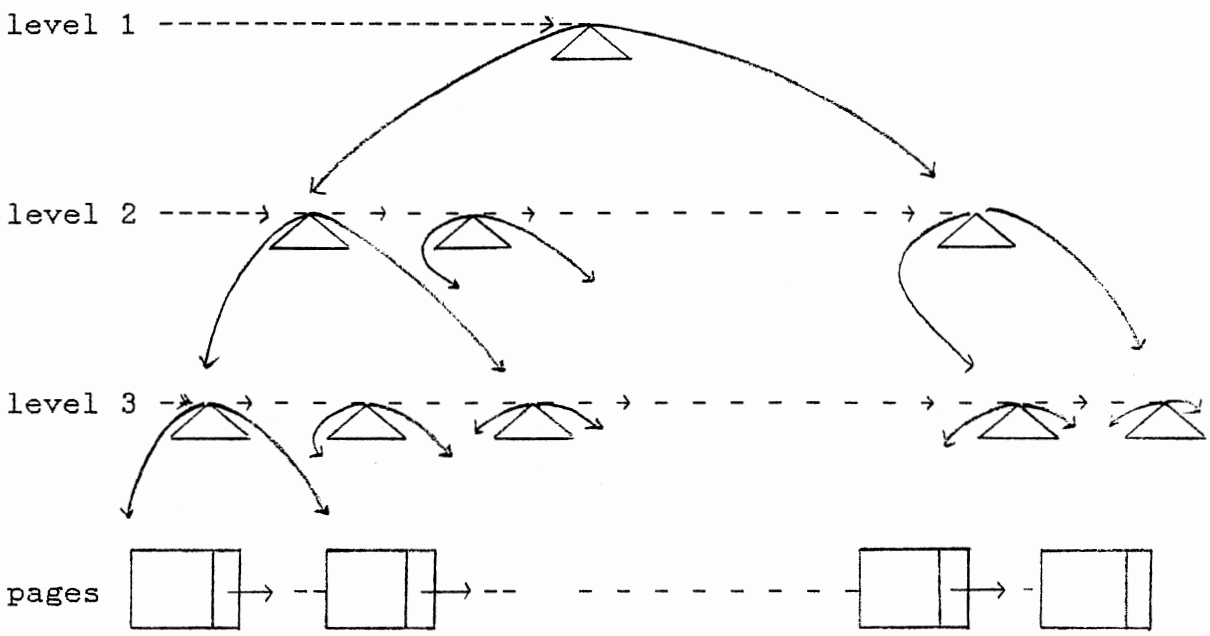


Figure-4. A multidimensional B-tree of 3 attributes.

a_i ". Finally, we insert values $a_{i+2}, \ldots,$ an each corresponding to a single node B-tree.

A multidimensional B-tree is a fairly good dynamic structure for an environment with 'large' filial set sizes and complicated queries[30]. However, this structure is not efficient when it has very small filial set sizes especially on the last level.

Address computation technique

Conventional hashing is a technique for organizing direct access data structures with O(1) access time if there are no collisions. However, the access performance may be degraded

when collisions of records occur. Hashing is considered to be better than a tree structure organization in terms of average access time. Traditional hashing methods have two disadvantages over tree structures. First, hashing methods can not support sequential processing because a hash function scatters data over the entire data space destroying sequentiality in the original data. Second, hash tables are not extendible and their size is intimately tied to the hash function used. So if we use improper hash function with a low estimate of the size of data set to be processed, a complete reorganization of the hash table may be required. Thus, conventional hashing methods are good for static data sets.

Several hashing methods of dynamic form were introduced in late 1970's such as dynamic hashing[20], extendible hashing[11] and linear hashing[23]. These are all single key hash schemes. We review here two multidimensional versions of extendible- and linear- hashing: the extendible cell method[39] and interpolation based index maintenance[6].

The extendible cell method[39] is a two dimensional version of (one dimensional) extendible hashing. We describe first the extendible hashing scheme and then generalize it to a multi- dimensional hashing. Access time is the most important performance characteristic of a hashing scheme. Dynamic characteristics which are lacking in conventional hashing can be obtained by interposing a large directory address space between key space and the physical address space. An extendible hashing file is structured into two parts: a directory

and leaf pages (buckets). The directory usually has a header in which is stored a quantity called the depth of the directory. The directory of the file is a linear table with 2^{dx} elements. At depth dx of the directory the hash function distributes the point data of file F onto an even interval with x-spacing 2^{-dx} which accommodates 2^{dx} pointers to leaf pages. We may reorganize the directory at each doubling without affecting the leaf pages. We can assume the data set domain D to be [0,1). Let x in D have the binary representation $x = \Sigma\, a_i 2^{-i}$ and f be the hash function f(x) which generates the binary representation. This scheme can be implemented by an array index calculation using $idx(x,dx) = \lfloor 2^{dx}\, x \rfloor$ so that the directory forms a one dimensional array of size 2^{dx}. The entry of the directory is to be an address pointing to a leaf page. Each leaf page in which point data are stored has a header that contains a local depth d'x for the leaf page such that dx >= d'x. The relation dx >= d'x means that the leaf page of d'x is pointed to by more than one directory pointer. Overflow in this leaf page does not necessarily trigger directory doubling.

Extendible cell method(EXCELL)[39] is an adaptation of this hashing scheme to two dimensional data space. We assume the data set domain to be the unit square D = [0,1) x [0,1). Let (x,y) in D have the binary representation such as $x = \Sigma_{i>=1}\, a_i 2^{-i}$, $y = \Sigma_{i>=1}\, b_i 2^{-i}$ and g be the following hash function $g(x,y) = \Sigma(a_i 2^{-(2i-1)} + b_i 2^{-2i})$. By definition an EXCELL implementation of a point file F on D is the structure

obtained by applying extendible hashing on the one dimensional interval [0,1) to g(F). But it is not simple to implement efficiently the function g. However, it implies that the leaf intervals of EXCELL correspond to rectangles formed by halving the domain alternatingly in the x- and y-directions, So an EXCELL file maintains a directory which is extendible without affecting the leaf buckets. Replacing the hashing function g by an array pointer calculation such as idx(x,y) = $2^{dy} \lfloor 2^{dx} x \rfloor + \lfloor 2^{dy} y \rfloor$, we can have the relation order of a directory entry so that the directory forms a two dimensional array. In this case, the depth d is dx + dy where dx = $\lceil d/2 \rceil$ and dy = $\lfloor d/2 \rfloor$. An example of a directory in an EXCELL method is illustrated pictorially in Figure-5 in case of depth 3 and d'x = 2, d'y = 1 and bucket capacity is 2.

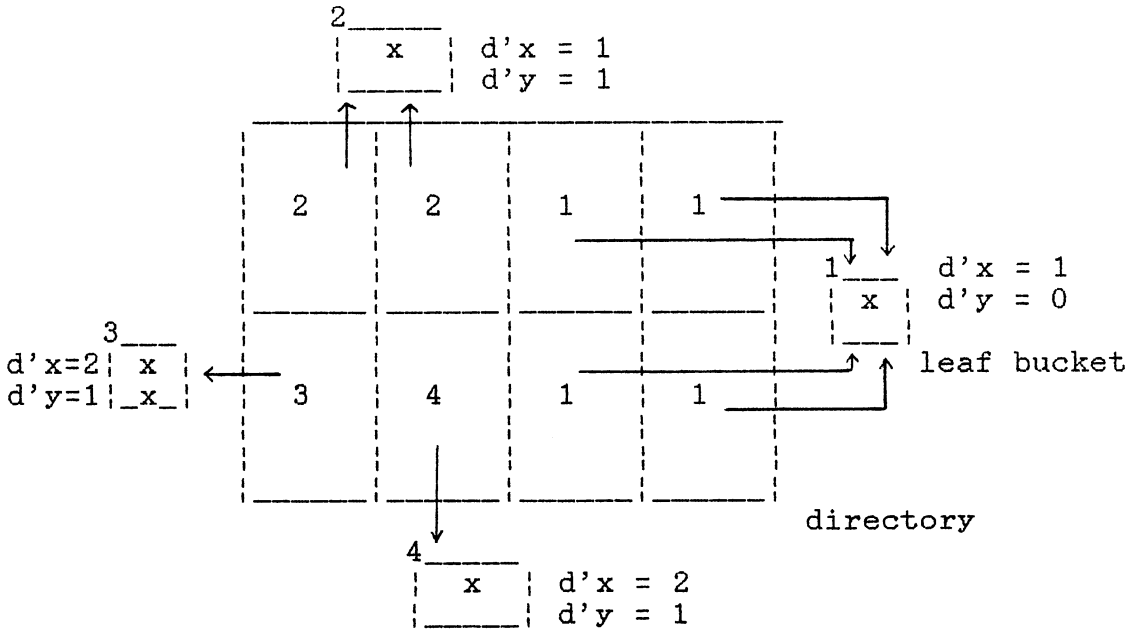


Figure-5. Directory maintenance in an EXCELL method.

Insertion into directory space 3 in Figure-5 results in an overflow in the corresponding bucket. At that moment we split the bucket and double the directory. The doubled directory is given in Figure-6. Deletion can be implemented in reverse way checking underflow and merging bucket or directory partition when it is necessary.

The directory is an array of elements which corresponds to a rectangle of minimal equal size. To access a record, we use the value of its attribute to determine an entry to the directory array using the formula for idx(x,y). The entry value points to the bucket in which the record is stored. This method requires no more than two disk access for a retrieval of a point data and is suitable for dynamically varying sets of

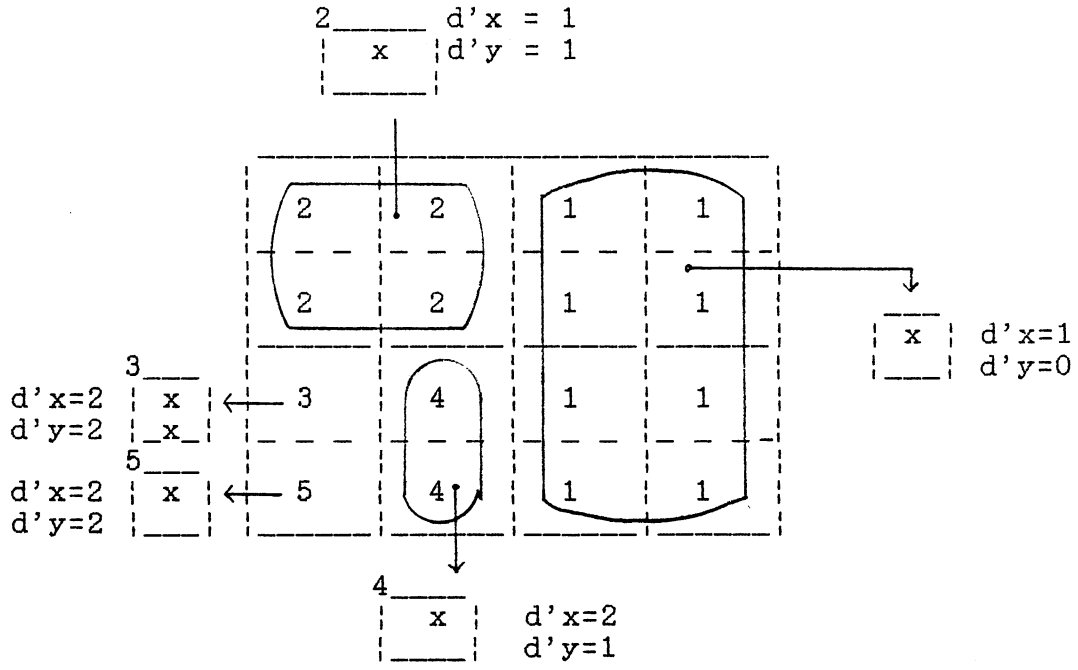


Figure-6. Directory doubling in an EXCELL method.

data with successive allocations of memory. Only when overflow occurs in a leaf page pointed to by an entry of the smallest cell in the directory, one must double the size of the directory. This scheme can be efficient for uniformly distributed data sets such as geometric information.

The second hashing method considered is the interpolation based index maintenance[6] which is based on linear hashing [23]. It extends the classical hash file organization using chaining for overflow areas. A chain is an explicit linear list of pages, the first page of which is fixed sized primary page and all subsequent pages in a chain are fixed sized overflow pages. This method supports common operations such as insert, delete, update and find.

Using an example, we can easily capture the concept underlying linear hashing. Suppose that h_0 is the hashing function used to insert the records into a file F. Let $k' = h_0(k)$ be the index of the chain that must contain the record. We insert records with the keys:

12, 10, 8, 5, 7, 9, 20, 26, 13, 25

where $h_0(x) = x \bmod 3$, primary page size = 3, overflow page size = 2. We see the resulting set of 3 chains after inserting above records in that order in Figure-7 . The storage utilization factor L is defined to be the ratio of the number of records in the file to the number of available records in the existing chains. Then in this example L = 10/13 = 0.769.

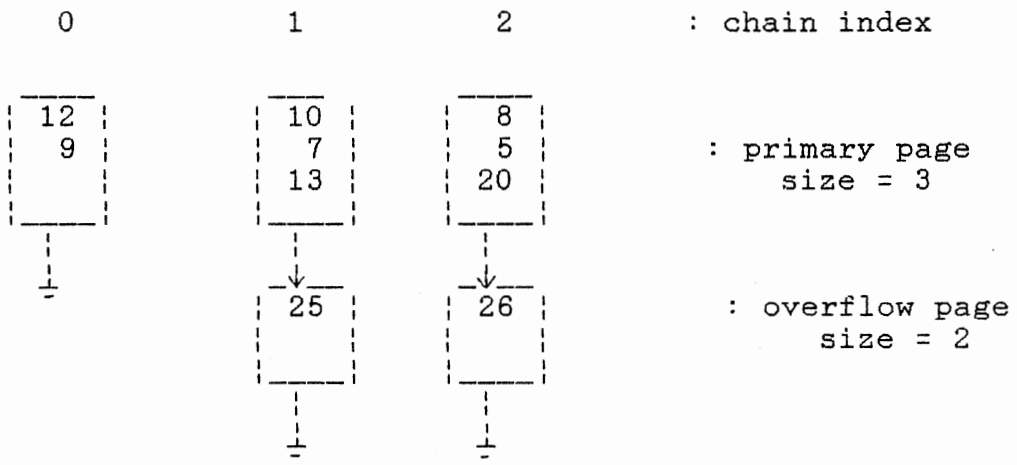


Figure-7. Chain representation in linear hashing.

Linear hash must specify upper and lower limit of the factor L, i.e. 0 <= a <= L <= b <= 1. Let a = 0.40 and b = 0.80. If we insert another record '16' into the example file, L becomes 11/13 = 0.846 which is over the upper limit b. Then we adopt split operation on a chain sequentially by chain index from the first chain, which creates a new chain and distribute the records into two chains using a new hash function. In our example, we split chain 0 using a new hash function $h_1(x) = x \bmod 6$. Figure-8 shows the result with L = 11/16 = 0.6875. As more and more records are inserted into the file, we inevitably meet the situation in which every original chain has been split, thus appearing as if it were loaded using hash function h_1. To locate a record with key value k, we should check whether the chain obtained using $k' = h_0(k)$ has split or not. If it is split, we access chain $k' = h_1(k)$.

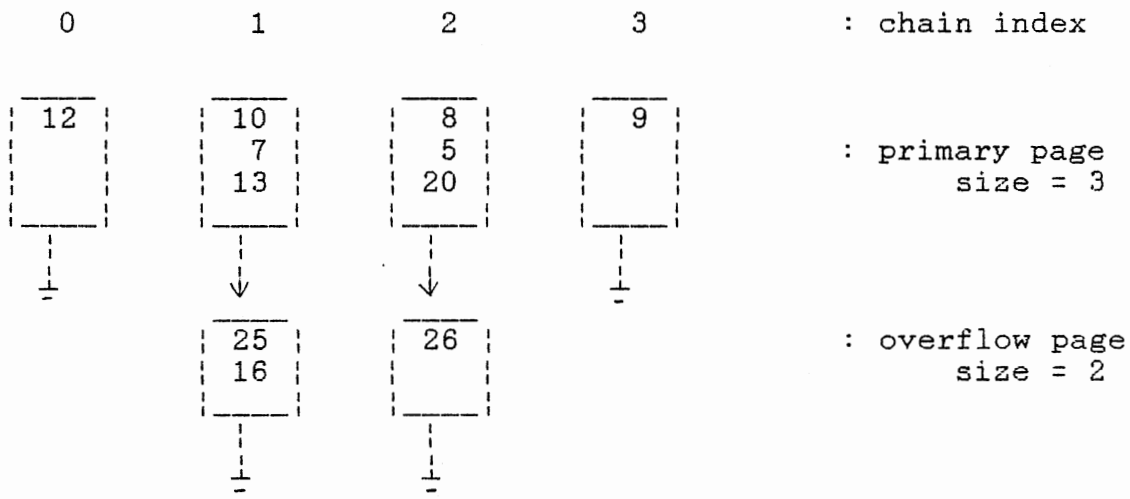


Figure-8. Chain split in linear hashing.

Interpolation-based index maintenance[6] is a multi-dimensional application of linear hashing. Suppose that the key space D consists of elements in the k-dimensional space $[0,1)^k$. For the purpose of utilizing linear hashing, we define a linear order , S(a), mapping from d= $[0,1)^k$ to [0,1). Let a = (a_1 , a_2 ,..., a_k) be a point datum in D. Each component a_j , $1<=j<=k$, of the point a has a binary representation.

$$a_j = \sum_{i>=1} a_{ji}\, 2^{-i}$$

Then define

$$S(a) = \sum_{i>=1} \sum_{1<=j<=k} a_{ji}\, 2^{-k(i-1)-j}, \quad k: \text{constant}$$

S(a) now denote a single key which is obtained by interleaving bits of original k keys. Hence we can use this single key for linear hashing with an appropriate sequence of hash functions $H = h_0\ h_1\ \ldots$

In contrast to extendible hashing method, this scheme does not maintain a directory for accessing the record page. So we can expect average successful search length of much less than two. This scheme utilizes overflow areas by chaining. Therefore, this structure can have worst-case problem of taking O(n) steps in operations for a file with n records.

CHAPTER III

GRID FILE

Introduction

The Grid File is one of the grid method structures that organizes the data space in which those data are embedded. Its major design goal is to retrieve records with at most two disk accesses from a large volume of data. This scheme maintains a grid directory which performs mapping of grid blocks to data buckets. All records in one grid block are stored in the same bucket. Several grid blocks can share a bucket as long as the union of these grid blocks forms a k-dimensional rectangle in the record space. The grid directory is used to keep a dynamic correspondence between the grid block and the data buckets. The grid directory consists of two parts: k one-dimensional arrays called linear scales and a dynamic k-dimensional array called the grid array(we will use grid directory for grid array). Linear scales define a partition of the domain of each attribute and are used for computing grid block addresses. The linear scales are kept in primary memory and support the operations to be defined on a grid file structure in the following section. The grids defined by linear scales are in one-to-one correspondence with the blocks of a grid directory. The values of the elements of the directory are pointers to the

relevant data buckets. The grid directory grows easily so large that most of its elements should be kept on disk during processing since we are handling 'large' files. Buckets which usually have more than ten records are kept on disk. The size of a bucket is usually a fixed unit of physical transfer, a page.

As a dynamic structure, a grid file supports insertions and deletions on-line. Maintaining grid directory dynamically is the heart of a grid file structure. When buckets overflow as more and more records are inserted, a split-operation is triggered. There are two types of splitting. The first, which is more common, occurs when several grid blocks share a bucket that has just overflowed. In this case, we need only to get a new bucket, distribute data between the old and new buckets and adjust the mapping between grid blocks and buckets. The second type arises when we must refine the embedding space(grid directory) in addition to the first type of splitting. This is caused by an overflow in a bucket, all of whose records lie in a single grid block. The merging process has also two types: bucket merging and directory merging. Bucket merging, which is more common, occurs when the occupancy of a pair of adjacent buckets is under a certain threshold. Directory merging arises when two adjacent cross sections in the grid directory have identical values. This type is rarely of interest except when the file shrinks continuously.

Nievergelt et al.[27] specified in their grid file design only those decisions which seem to be essential such as:

- grid partitions of search space,
- assignments of grid blocks to buckets that result in convex bucket regions,
- grid directory, consisting of a large dynamic array but small linear scales.

There remain several design policies open because those can be established in many different ways by each implementor. The most important open issues are (1) choice of splitting policy, (2) choice of merging policy, (3) implementation of the grid directory, and (4) concurrent access. An efficient implementation of the open strategies are the major objectives of this thesis.

Abstract data types underlying the grid file

A grid file essentially consists of three simple abstract data types: linear scales, a grid directory and data buckets. Linear scales are k one-dimensional arrays, the elements of which represent boundaries of intervals in each dimension. The grid directory is a k-dimensional dynamic array whose elements are pointers to data buckets. Data buckets are fixed sized structures of records in which data are stored. Data structures and operations associated with the abstract data types can be described as follows in k-dimensional data space, S_k:

$$S_k = D_1 \times D_2 \times \ldots \times D_k .$$

1) Linear scale

A grid file partitions the data space into orthogonal grids. The k scales define intervals in each dimension of the data space. For a grid file of k dimensions, we can characterize its data structure as follow.

Linear scale: s_i[no_of_boundary] of scaleType
scaleType can be one of integer, real character, or string.
$1 <= i <= k$, i,k : integer

Basic operations defined on scales are as follows.

INDEX(s_i, key_i, no_of_boundary): This procedure finds interval index value 'idx' of key_i in scale s_i.

SPLIT_SCALE(s_i, b_i): This procedure inserts a new boundary, b_i, into scale s_i.

MERGE_SCALE(s_i, b_i): This procedure deletes a boundary b_i from scale s_i.

In addition, several binary buddy operations to be discussed in the following chapters are supported by scales.

2) Grid directory

A grid directory is introduced to represent and maintain the dynamic correspondence between grid blocks in the data space and data bucket. This is a k-dimensional dynamic array. The specific data structure of a grid directory is implemen-

tation dependent. At the moment, we define grid directory formally as a conventional array as follows.

Grid directory : DR[0:n_1-1][0:n_2-1]...[0:n_k-1],

n_i :integer

The procedures on a grid directory are defined as follows.

ACCESS(DR,r,p): This procedure finds a pointer value p in directory DR. 'p' is an address of a data bucket which contains record 'r'.

NEXT_BELOW(DR,D_i): This procedure returns the neighbor element of current block in a grid directory, DR, to "below" direction in dimension D_i.

NEXT_ABOVE(DR,D_i): This procedure returns the neighbor element of current block in a grid directory , DR, to "above" direction in dimension D_i.

SPLIT_DR(DR,idx,D_i): This procedure splits directory DR at interval 'idx' in a dimension D_i. Given idx, create a new element idx+1 and rename all grid blocks above idx.

MERGE_DR(DR,idx,D_i): This procedure merges directory DR at interval 'idx' in dimension D_i. Given idx, remove a row or column of idx-1 and rename all grid block above idx.

ASSIGN(DR,p,B): This procedure assign a grid block with p to data bucket B.

3) Data Bucket

A data bucket is a fixed sized structure of records and some additional information such as record count. The data structure used to organize records within a bucket is of minor importance in a grid file structure as a whole. The data structure of a bucket can be declared as follows.

```
Bucket structure {
                    Int count,
                    Char Record[],
                    Char NonkeyInfo[],
                  };
```

The operations defined on the bucket are as follows.

SPLIT_BK(B_1, B_2): This procedure allocates a new data bucket B_2 and distributes records in B_1 into the two buckets. Update the included additional information if it is used.

MERGE_BK(B_1, B_2): This procedure moves records in bucket B_2 to bucket B_1 and frees bucket B_2. Update the included additional information in bucket B_1 if it is used.

Based on the above data structures and operations on them, a grid file supports the common operations in file structures such as FIND, INSERT, DELETE, UPDATE and some RANGE_QUERY. The operations on a grid file F can be defined as follows.

FIND(F,r,B): This procedure searches the file F to find

the record 'r' and returns the record position if record 'r' is found or -1 if R is not found in the file F.

INSERT(F,r): This procedure inserts a record 'r' into file F if 'r' is not found in F.

DELETE(F,r): This procedure deletes record 'r' from file F if 'r' is found in F.

UPDATE(F,r): This procedure updates information of record 'r' in file F.

RANGE_QUERY(F,range): This procedure reports all records in file F satisfying the given range.

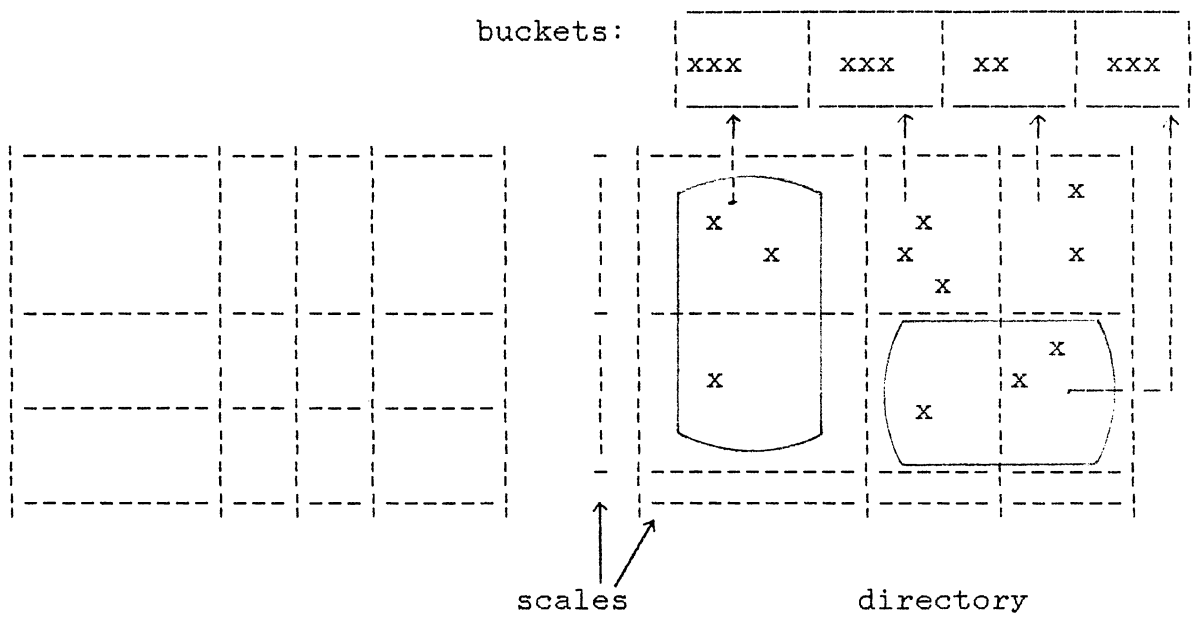
The performance of insertions and deletions in a grid file are likely to be influenced by efficiency of SPLIT_() and MERGE_() operations. The operations are again highly dependent on the specific implementation of grid directory. The concepts and strategies with this problem are discussed in Chapter IV. The associated algorithms are established in Chapter V.

CHAPTER IV

CONCEPT AND STRATEGIES FOR IMPLEMENTATION OF GRID FILE

Organization of grid file

The grid file, as a large data structure based on the technique that organizes the embedding space from which the data are drawn, decomposes the data space as shown in Figure-9a. Every subdivision of the existing grid space subdivides it into two subspaces. 'Linear scales' are used to keep track of the boundaries of the respective subdivisions. The grid partition of the data space directly correspond to a k-dimensional array called a 'grid directory'. A cell in the partition is called a 'grid block'. Records are stored in a fixed-sized storage unit on disk, which is usually called a 'bucket'. Several grid blocks may share a bucket for memory optimization. Such a set of grid blocks forms a 'bucket region'. Actually we may need only the structure to organize the set of buckets. However, by maintaining the set of buckets and grid blocks separately, while keeping a certain correspondence between them, we can design a more efficient dynamic structure. The grid directory is introduced to keep such a correspondence. In Figure-9b, we describe the above relations in the case of 2-dimensions.



(a) Grid partition (b) Grid file organization

Figure-9. Grid partition and grid file organization

We usually evaluate the efficiency of a large search structure by disk access time, update time and memory utilization. The basic design objectives of a grid file are:

(1) no more than two disk accesses for point queries;

(2) splitting and merging of grid blocks to involve only two buckets;

(3) efficient processing of range queries in large linearly ordered domains;

(4) maintaining a reasonable lower bound on average bucket occupancy.

We establish several strategies to be imposed on the

operations discussed in Chapter III just based on the above design objectives.

Access time bound

As described in section (2) of Chapter III, a grid directory is originally defined as a dynamic k-dimensional array. The basic operations on grid directory and scales such as ACCESS, NEXT_BELOW OR NEXT_ABOVE, SPLIT_, MERGE_, AND ASSIGN are also defined in Chapter III.

The mechanism of search operations is described with the above procedures in Figure-10. Assume that linear scales X and Y for a 2-d grid file attained the values indicated.

```
scale  X = ( 0, 20, 30, 40, 50, 70 )     :  Age
scale  Y = ( 5, 10, 15, 25, 40 )         :  Income $1,000 unit
```

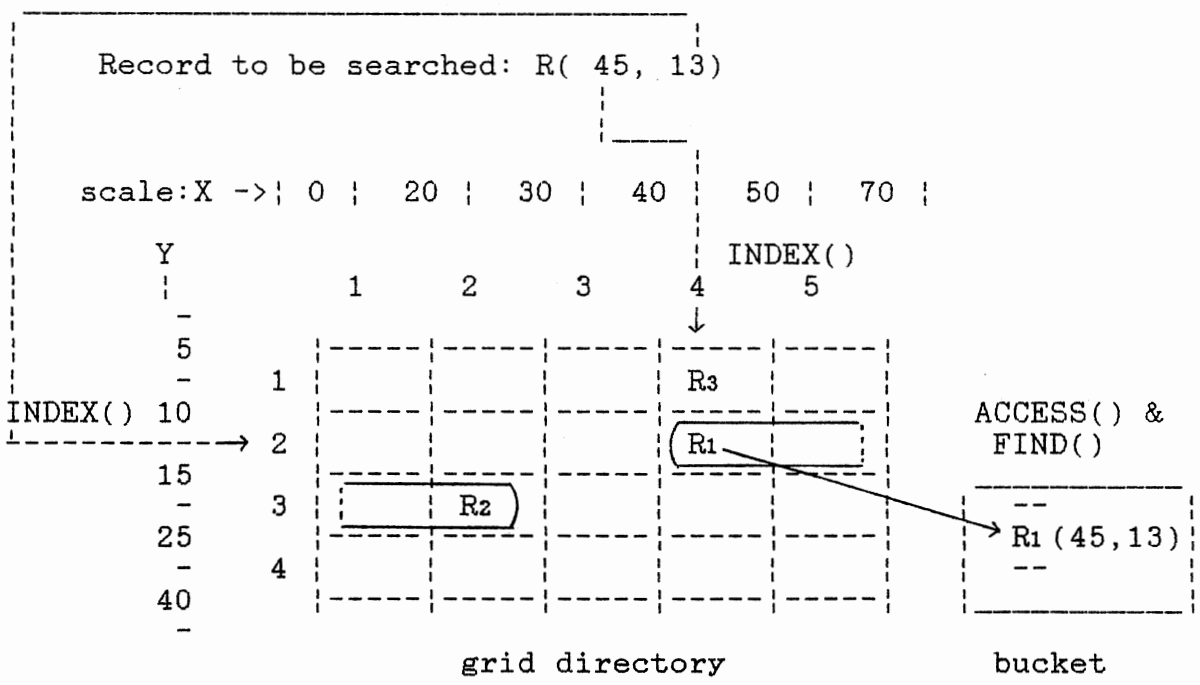


Figure-10. Record search mechanism

The grid directory is likely to be so large that it must be kept on disk. However, since scales X and Y are small enough to be kept in primary memory, INDEX() does not require a disk access to get the index values, 4 and 2, in scales X and Y respectively. With these values, we can compute the address of the corresponding block in the directory(See Appendix A). So we can read correct directory page in one disk access. With one more disk access to the bucket where the record r(45,13) resides, we can retrieve the record in two disk access times. Of course, we can search a record with one disk access if the directory entry corresponding to the record is in primary memory. We see the case of one disk access for searching when we search record r_2 immediately after r_1 assuming that a unit of disk access is four directory blocks. A bucket usually has a page size that can be read with one disk access. The organization of records within buckets is not an immediate interest in a grid file structure. In our design, records are fixed sized and are written entry-sequentially; i.e., we insert them by the sequence of their arrivals.

Resident grid directory

The above searching method implies that the grid directory is implemented as a conventional row major order array. Generally a conventional array allocation is recommended as an actual data structure for a grid directory because of its simple implementation, fast access and memory optimization[17]. How ever, we can expect some performance degradation when

neighborhood relation in all dimensions are important as in a geometric information system. We see that we can not retrieve the two neighbors, r_1 and r_3, at one time if one disk access reads only four disk blocks as assumed before. For efficient neighborhood operations in all dimensions , it is required that the elements of a directory for neighbors be stored in the same disk block if possible. As a solution to this problem, a 'resident grid directory' is suggested in [15]. A resident grid directory is a scaled down version of the grid directory, in which the limit of resolution is coarser. Figure-11 shows the relations among the resident grid directory, grid directory and actual bucket area.

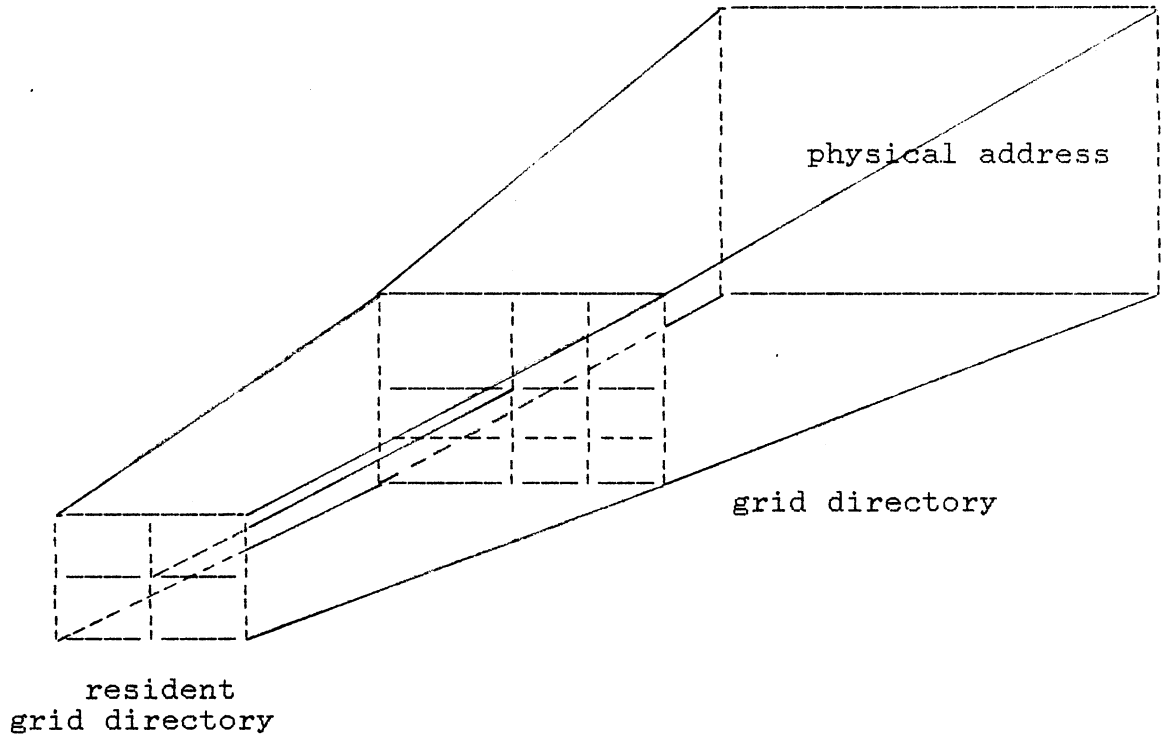


Figure-11. A double level grid directory representation.

In grid file design, it is assumed that grid directory is itself so large that most of its elements must be kept on secondary storage during data processing. The resident directory scheme partitions the directory space into grid form just as data space has grid partition. An element of resident directory is to point the corresponding directory block in secondary storage. The scales that are needed to contain the boundaries which indicate the grid partitions in resident directory are called 'resident scales'. The partial directory corresponding to each grid block of a resident directory is called a 'block directory'. The boundaries defining the grid partition in a block directory are kept in block scales. We keep only the resident directory and the resident scales in primary memory. A block directory is contained in a fixed-sized bucket located in secondary storage. It may be most reasonable that the corresponding block scales be contained in the same bucket. The bucket is called a 'directory bucket'. The size of a block directory is variable within a directory bucket. A directory bucket splits when it overflows as data buckets do. Several grid blocks in resident directory can also share a directory bucket with a restriction that the union of these grid blocks makes a convex form. The organization of resident directory scheme is shown in Figure-12. This scheme implies that grid directory is again implemented as a grid file with an assumption that the size of resident grid directory is small enough to be kept in primary memory. This assumption for preserving the time bound of two disk accesses is reasonable in

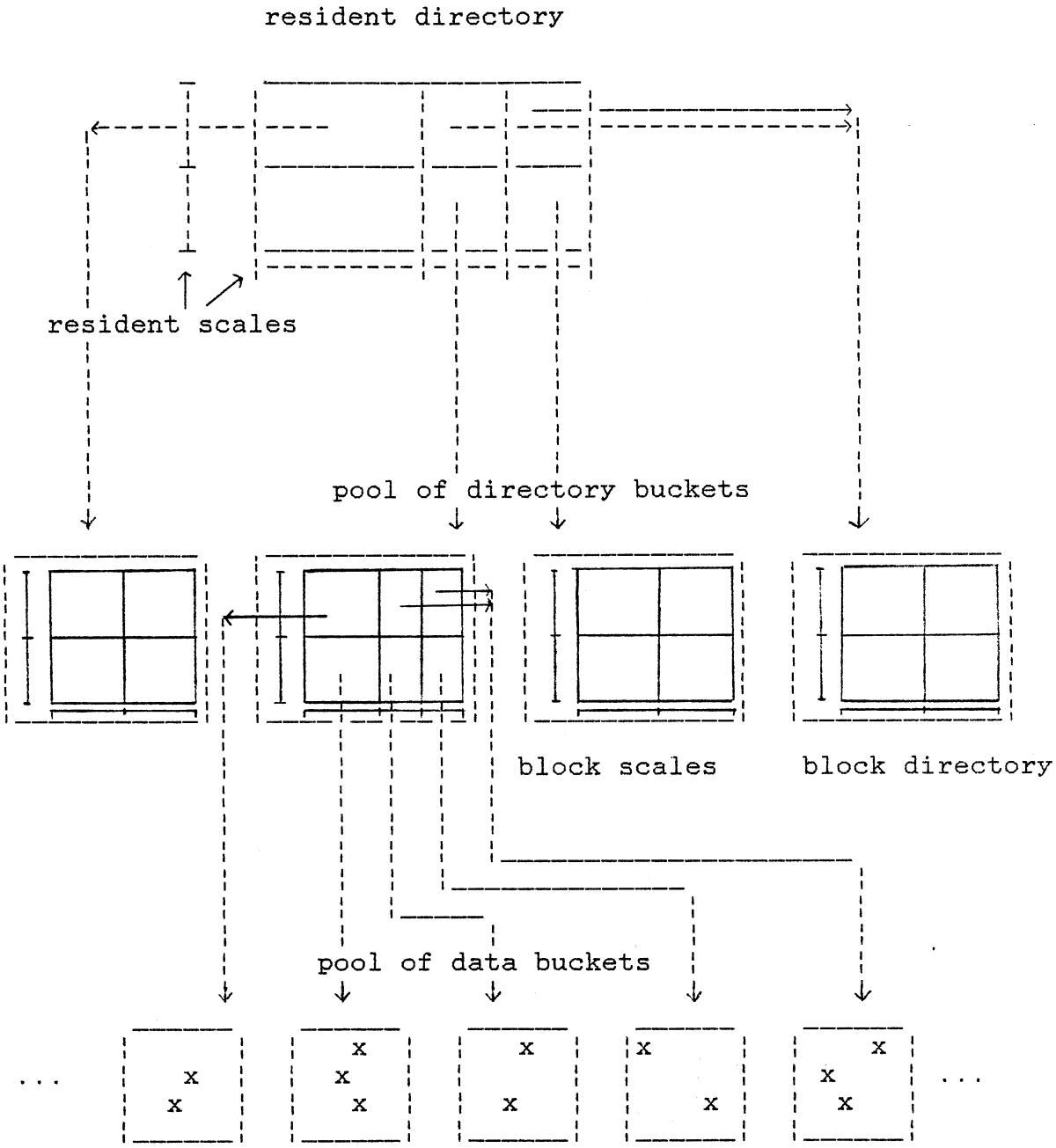


Figure-12. Organization of grid file with resident directory scheme.

most practical applications of the grid file(See Chapter I).

Assignment of grid blocks to buckets

The correspondence between grid blocks and buckets is maintained dynamically in the grid directory. We should decide reasonable strategies in assigning grid blocks to data buckets. In order to obtain one of the design objectives of a grid file structure, the upper bound of two disk accesses, we should guarantee that all the records in one grid block be stored in the same bucket. In contrast, several grid blocks must be able to share a bucket to keep a reasonable average bucket occupancy. We have already defined the bucket region as the grid blocks sharing a bucket. It is clear that the shape of bucket regions may affect the performance of range queries (discussed in Chapter VI) and mapping operations of directory after splitting and merging. Since the grid file system is based on grid partition of the data space as discussed in Chapter III, it is necessary to keep the bucket regions as a convex shape, a k-dimensional rectangle, in order to get high performance in the above operations. This convex assignment strategy of grid blocks to buckets is maintained in all of the operations defined on grid file structures. We can see an example of convex assignments of disk blocks to buckets with maximum record counts of two in Figure-13. This intermediate state during grid file processing is used as the basis for explaining splitting and merging strategies in the following sections. In Figure-13, 1' values denote the history of

splitting of corresponding buckets, which is discussed in the following section.

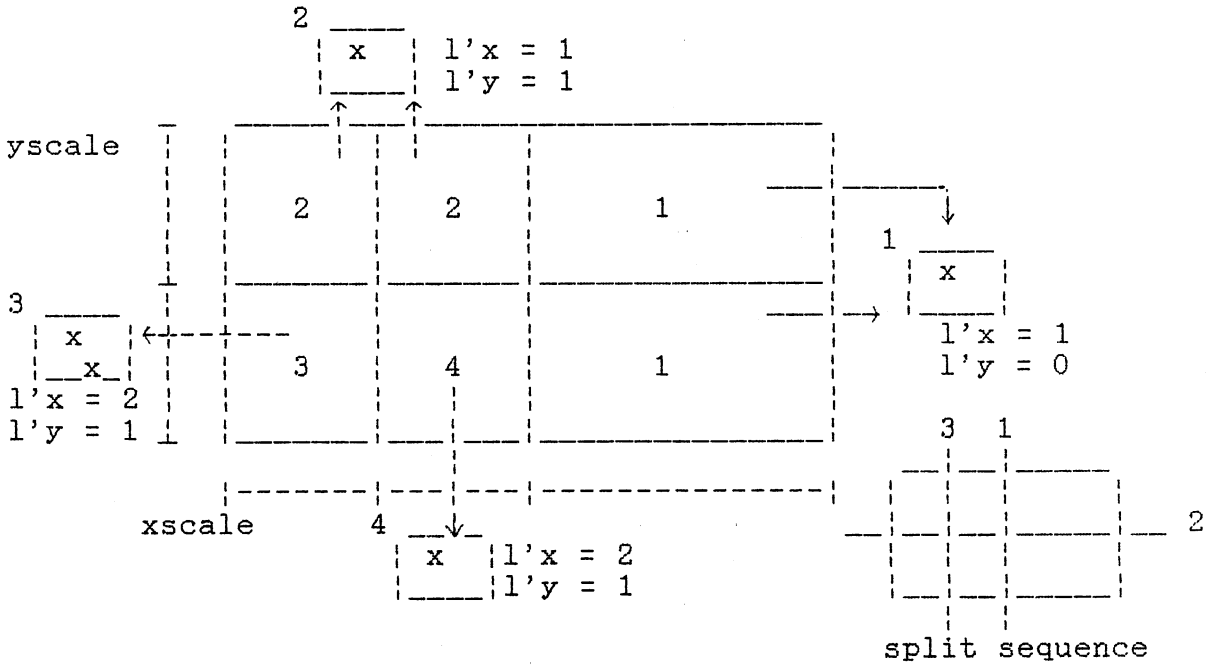


Figure-13. Assignment of grid blocks to buckets.

Splitting strategies

There are two types of splitting: (1) splitting only data buckets and (2) splitting data buckets accompanied by splitting of associated directory and scales. The first type of splitting occurs when an overflowed data bucket is shared by more than one grid block. This type has also two cases: one case is that the grid blocks sharing the data bucket to be split are adjacent in only one dimension, and the other case

is that they are adjacent in more than one dimension. In the previous case as bucket #2 in Figure-13, we merely create a new bucket at first. Then we move records according to the boundary value of grid blocks and adjust mapping of directory to the data bucket. In the latter case as in bucket #1 in Figure-13, we additionally need to decide the dimension on which base the records move between the two buckets involved in splitting. The second type of splitting is triggered when a overflowed data bucket is pointed to by a single grid block as in bucket #3 and #4 in Figure-13. For this kind of splitting, we must decide which dimension should be split and where in the selected dimension the new boundary should be inserted.

In choosing the dimension, we can adopt a 'cyclic' sequence among the dimensions according to a fixed schedule. A splitting policy may favor some attributes by splitting the corresponding dimensions more often than others if the characteristics of the data set are known. This results in a higher resolution in that dimension. One of our design objectives is that split operations involve only two data buckets, the original one to be split and a new one. It requires that a reasonable average bucket occupancy be maintained for effective memory utilization. This means that in the process of splitting a directory triggered by the split of bucket #3 we need not necessarily split the other bucket of which region is split as in the bucket #4 in Figure-13. The resulting state of the directory split is shown in Figure-14. In this regard, we need only one split boundary in the selected dimension to be

inserted in corresponding scale. The boundary need not necessarily be chosen at the middle point of the interval. In choosing a split boundary, there may be several methods such as binary buddy system , Fibonacci buddy system and weighted buddy system.

In our implementation, we prefer to preserve the same resolution in every dimension and try to make grid blocks have uniform shape of 'k_cubicle' as much as possible. With this regard, we impose higher priority on the dimension which has less split history in choosing split dimension. We choose the binary buddy system to split a boundary value which selects a new boundary by bisecting a region to be split. We discuss these policies more in Appendix C. The split history of a bucket, the number of split operations on the bucket, is maintained as a 'region level'(see l' values in Figure_13 and 14). We also maintain a value 'local level' which implies the split history of an interval in scales. These are discussed in Chapter V using binary buddy algorithm. The dimension with small value of region level has higher priority in splitting. The middle value of the region becomes a new boundary, which is inserted in the scale corresponding the dimension.

The simple bisecting of a region may incur bucket and directory splitting repeatedly without any moving of records to a new bucket if the data in the bucket to be split has clustered to the cross section of its region boundaries. Since in our grid file design we assume that the data set to be processed is 'large' and somewhat uniform distribution in the

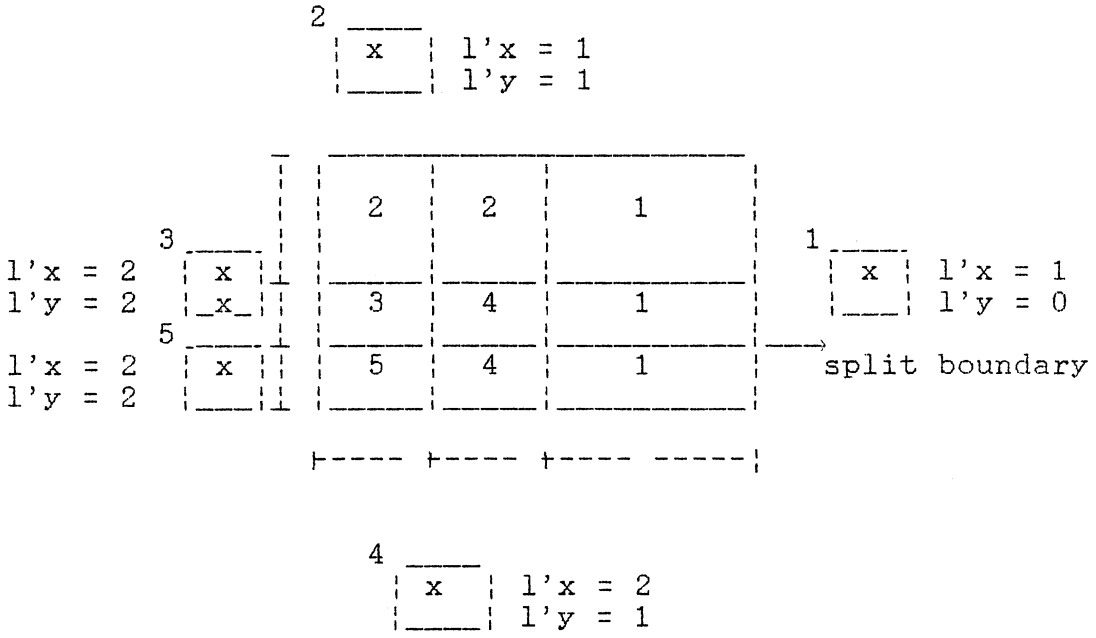


Figure-14. Splitting of bucket and directory

whole data space at its ultimate steady state, the split strategies chosen have their own right. We note that the directory size is a function of both the data volume and its distribution to data spaces based on grid partitioning. It is independent of the insertion sequence of a given data set.

In addition to the fundamental splitting operations in single level grid file structure discussed so far, we need to establish a certain constraint on resident directory level since we are going to implement a double level grid file structure. Splitting of resident directory and resident scales can be performed in exactly the same way as that of single level scheme based on the same splitting policies. It happens when a directory bucket corresponding to a single block in

resident directory overflows(as directory bucket #2 and #3 in Figure-12). A continued splitting of a block directory with its block scales overflows the fixed-sized directory bucket which accommodates them. At that time we have to split the directory bucket. The bucket split is shown in Figure-15. We also maintain region level of each directory bucket in resident directory as we do in block directory for each data bucket. In splitting a directory bucket, the split dimension and split boundary are decided relative to the resident directory. We also adopt the same policies in choosing the split dimension and split boundary as those for splitting a data bucket. The splitting constraints of the binary buddy system on the whole

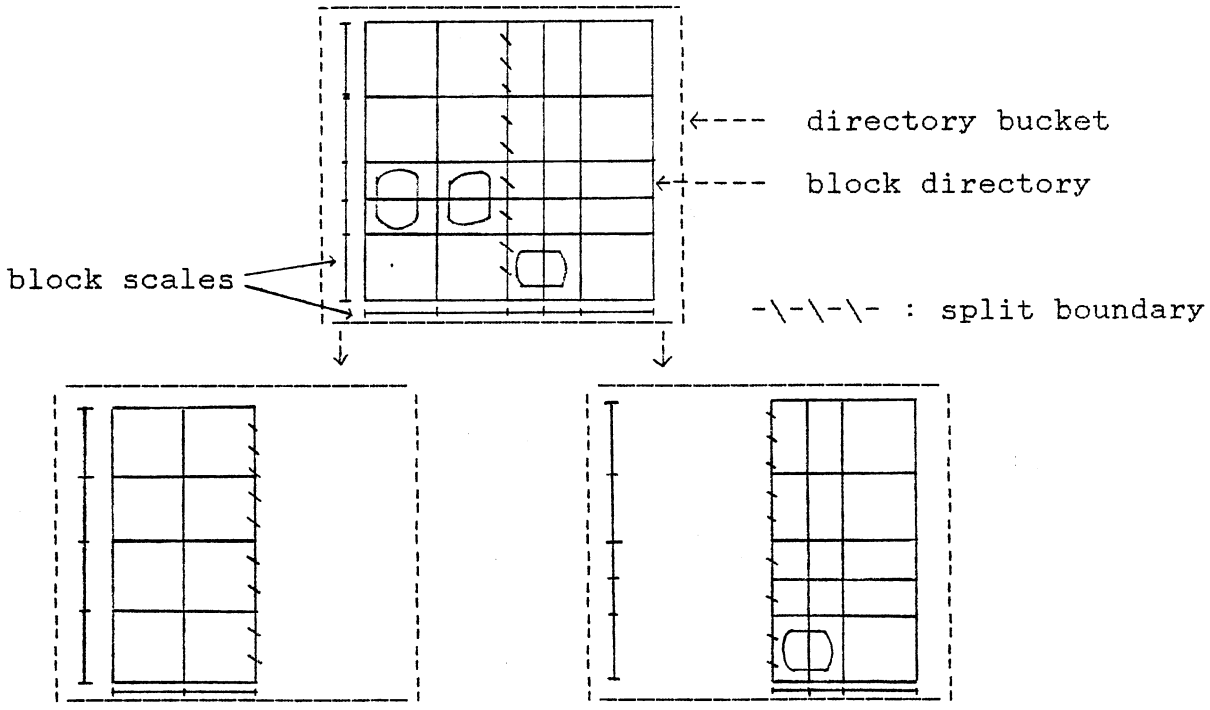


Figure-15. Splitting of a directory bucket.

data space guarantee that the split boundary on which the directory bucket is split always exists also as a boundary in the corresponding block scales. The block scales is divided at the boundary into two new block scales. Along the split boundary, the block directory is divided into two at the dimension defined by the scales. Each of two block directories and block scales is assigned separately to two new directory buckets and the other block scales not involved in the split are assigned to both of the new directory buckets. Finally we need to adjust the mapping between the resident directory and directory buckets. An example of grid file operations in Appendix B.

Merging strategies

Merging is attempted when a data bucket's occupancy falls below a certain lower threshold due to continued deletions. Actual merging is triggered when there is a proper candidate and the new bucket occupancy after merging would not be above a certain upper threshold. Bucket merging naturally makes a bucket region in a directory. Considering a bucket region corresponding to the bucket #1 in Figure-13, the vertical boundary inside the bucket region is unnecessary. We can also expect another situation for directory merging when buckets #3 and #4 are to be merged in Figure-13. However, considering the overhead of splitting and merging, directory merging is not always preferable in most applications especially when the file size is growing steadily or the file has reached a steady state

where the frequencies of insertions and deletions are almost same.

A bucket merging operation usually requires three decision policies: (1) selecting candidate buckets for merging; (2) if there are several candidates, determining which ones are to be merged; (3) setting the upper and lower thresholds of bucket occupancy after and before merging. For selecting candidates, two different methods have been suggested in [27]: namely, the neighbor system and the k-dimensional (binary) buddy system. Figure-16 shows the two methods. 'Buddy system' allows a data bucket to merge only one adjacent buddy in each dimension. Hence, the number of candidate bucket may be up to k. The 'neighbor system' has up to 2k candidates since it allows the data bucket to merge with either of its two adjacent neighbors provided that the resulting bucket region is also convex form. Since the number of candidates in neighbor system is large than that of buddy system, there may be more chances to merge bucket.

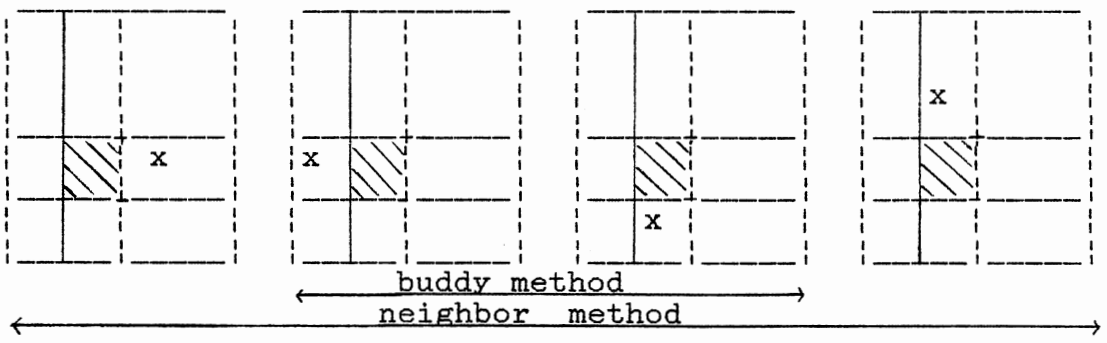


Figure-16. Buddy and neighbor methods in merging.

However, the merging by neighbor method may give rise to the 'no more mergeable state' condition. An example of this state appears in Figure-17. Though the buddy system in two dimensional grid file guarantees no occurrence of this state, it also possible in more than two dimensional grid file. This no-mergeable-state in directory clearly affects the grid file performance resulting in lower average data bucket occupancy and requiring more access of the directory in range queries because of large directory size. For lower and upper thresholds of bucket occupancy which trigger merging operations, there is no optimal levels obtained by mathematical analysis yet. Nievergelt et al.[27] suggest around 70 percent and no more than 80 percent value for upper threshold.

In our implementation of grid file, we also choose the binary buddy system for searching for a proper candidate for merging to maintain consistency with the splitting operation. In this system all buddies have same region levels in all corresponding dimensions, thereby assuring that the bucket region made after merging be convex form. If there is more than one candidate, we choose that which has largest value of region level to keep the directory block shape uniform since it has split more times. We choose 25 percent and 75 percent for lower and upper thresholds respectively.

In merging directory bucket, two block directories are merged into a bucket if the new directory bucket occupancy is sufficiently below the bucket capacity. Otherwise, the new directory bucket has to be split again with a few split of its

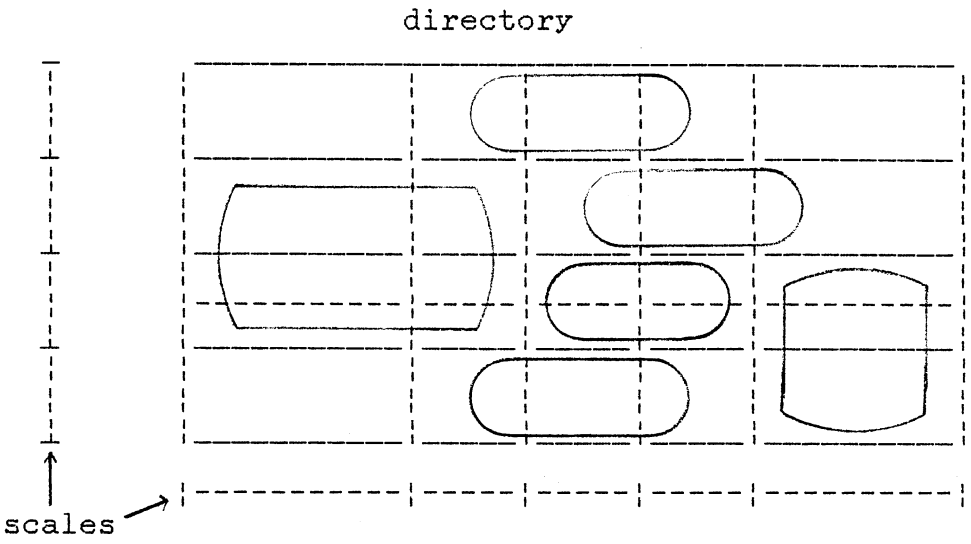


Figure-17. No more mergeable state of directory.

block directory and block scales. No optimum upper threshold is known yet. By one directory split operation, the directory size grows more greatly in higher directory bucket occupancy and in a higher dimension structure. Merging directory buckets also may be necessary only for continuously shrinking files considering the overhead of split and merge. Anyway, we can use the same methods for searching for candidate buckets as those used in merging data buckets. The two block scales of the bucket to be merged are appended to each other along the merge dimension. We can obtain all other block scales defining other dimensions for the new block directory from corresponding block scales in the old block directories by choosing more refined scales in every dimension. We see the merging of directory bucket in Figure-18.

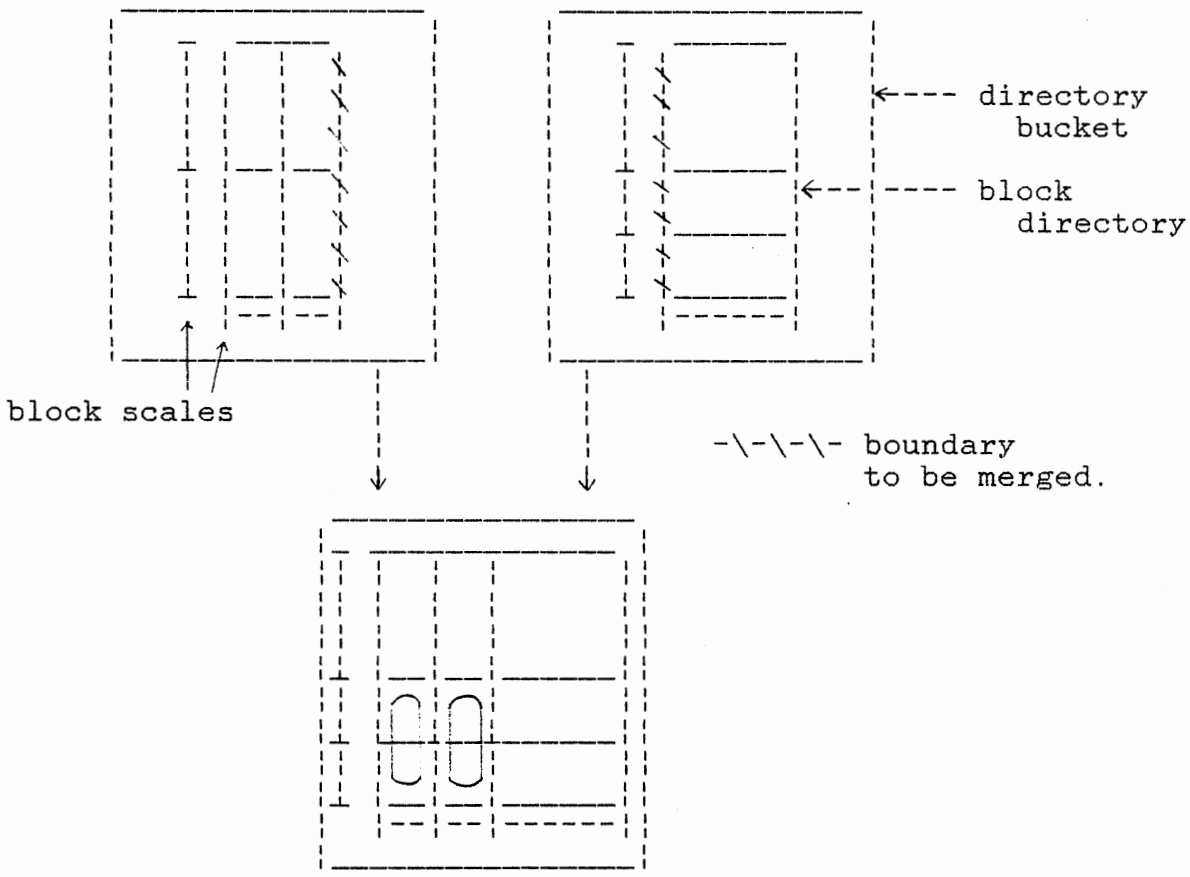


Figure-18. Merging of directory bucket.

CHAPTER V

ALGORITHMS FOR A GRID FILE

The algorithms for our implementation of a grid file are discussed in this chapter. In our design of a grid file, the grid scales are simple one dimensional dynamic arrays and the grid directory is a k-dimensional dynamic array of row major order. Splitting and merging operations are based on binary buddy system. The value 'level' of a bucket which implies the split history upon the bucket is maintained in corresponding grid blocks of the buckets. It is just the number of splitting operations to get the corresponding intervals in scales and is based on the similar concept of 'depth' in extendible hashing structure[39] discussed in Chapter II. We will call this a 'region level' to differentiate it from a 'local level' of a single grid block which is defined as an interval in scales.

The value of a region level is kept in the directory along with a pointer to a bucket and count of records in the bucket in our design. This means that we have most of the information about bucket in a directory rather than in the bucket itself. This may result in a large directory but we can gain efficiency of fewer disk accesses in deciding a dimension and a boundary value for splitting and merging operations. If a data bucket has the same value for region and local levels, it means that

the bucket corresponds to a single directory block and with several blocks which make a bucket region otherwise. Local level can be obtained by a simple computation on scales as they keep the boundary values of splitting operations.

Basic algorithms for binary buddy system

Our implementation of grid file is based on a binary buddy system in splitting and merging operations associated with insertion and deletion respectively. The history of repeated bucket splitting can be represented in the form of a binary tree. The tree representation with its grid file structure is shown in Figure-19, which shows the state after buckets #4 and #1 are split in Figure-14. The leaf values are pointers to bucket addresses in disk storage. Only the leaf values are kept in directory entries. Splitting occurs at any level. However, split of bucket on lowest level triggers directory split additionally(as in leaves #3, 4, 5, and 6). But merge can be done on lowest level only. In our algorithms, mergeable sets are (3,5), (4,6), (3,4) and (5,6) which make convex regions.

In order to simplify the explanation of the algorithms hereafter, we implement our grid file in the data space of long integer [0, MAX), MAX = 2^n, n = 16. The entries of grid scales are by definition boundary values of intervals in each dimension of the data space. Algorithm 1 obtains the local level of an interval. If we are given the lower and upper boundary of an interval in a scale based on binary buddy

system, the correctness of the algorithm is straightforward.

Algorithm 1

```
get_level(scale,low_bound,up_bound,MAX)

        i  <- 0;
        while( i < n )
            j <-  Right_shift( MAX, i);
            if ( j = (up_bound - low_bound) )
                return(i);
            fi
            i <- i + 1;
       end while
 end get_level
```

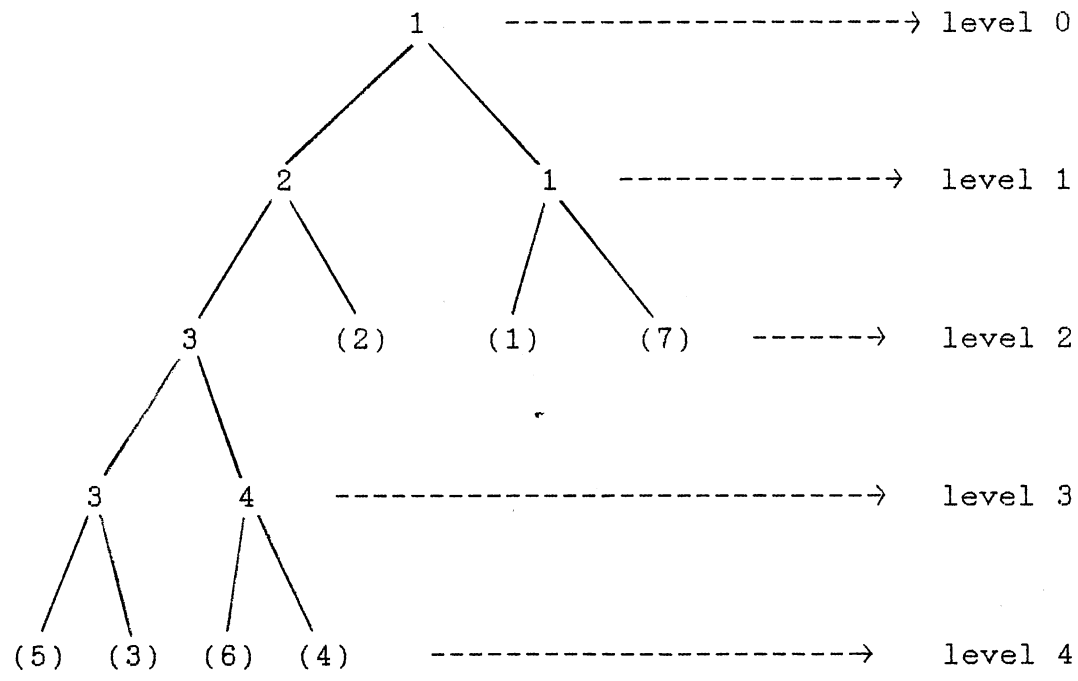


Figure-19. Tree representation of bucket splitting.

We need to know the buddy region of a given region for merging operation. In binary buddy system, the buddies have same levels in every dimension corresponding each other. There is only one valid buddy of an interval in each dimension between upper neighbor and lower neighbor which has same size of region. We can decide the valid buddy by checking the validity of the combined region of the buddies in view of binary buddy system because a pair of buddies should have been split from a region. The algorithm for getting buddy is shown below.

Algorithm 2

```
Is_valid_interval (scale,low,up,MAX)

   i <- right_shift(MAX, level);
   if(( low MOD i) = 0 )
        return(valid);
    else
        return(invalid);
   fi

end Is_valid_interval
```

Algorithm 3

```
 Get_buddy(scale, low, up, MAX)

   if( low = minimum value in scale)
      low_of_buddy <- up;
      up_of_buddy  <- up + (up - low);
   else if( Is_valid_interval(scale,low-(up-low),up,MAX)
      low_of_buddy <- low-(up-low);
      up_of_buddy  <- low;
   else
      low_of_buddy <- up;
      up_of_buddy  <- up + (up-low);
   fi
 return(low_of_buddy,up_of_buddy);

end Get_buddy
```

In order to rearrange the mapping between directory and buckets after insertions and deletions, we need sometimes to get the whole bucket region of a directory block which contains the records involved in the operations. With INDEX operation, we obtain an index value of an interval in a scale where a record is located. With the index value, we can get lower bound and upper bound of the interval in each dimension. We compute local level of the interval with the bound values. The region level of each dimension is kept in each directory entry. With the difference value of the two levels, we get its region boundary values. This procedure follows Algorithm 4 below. For example, we should rearrange mapping of the whole region of bucket #1 if a record is inserted into or deleted from the bucket.

Algorithm 4

```
Get_region(scale,low,up,region_level,MAX)

    local_level <- get_level(scale,low,up,MAX);
    i <- (local_level - region_level);
    call get_bud()  i times;
    return(whole buddy range);

end Get_region
```

Grid file algorithms

We have decomposed a grid file into three abstract data types in Chapter III,namely, grid scales, grid directory and data bucket, and defined procedures on the ADT's. Procedures

on grid directory and data buckets are dependent on those of grid scales. Maintaining scales in primary memory is a unique scheme of grid file structure. With indices obtained by the Algorithm 5, we can compute the address of a block in the directory corresponding to the data bucket(See Appendix A). The procedures on scales are performed by the following algorithms.

Algorithm 5

```
Index(scale, key, no_of_boundary)
   curr_index <- 0;
   while( curr_index < no_of_boundary)
      if( key <  scale[curr_index] )
            increment curr_index;
            return( curr_index);
      fi
      increment curr_index;
   end while
   return(unsuccess);
end Index


Split_scale(scale, curr_index, no_of_boundary)

   new_bound <- (middle value of lower bound and upper
                 bound of curr_index interval in the scale);
   allocate new_scale the size of which increased by 1;
   i, j <- 0;
   while ( i < no_of_boundary)
      new_scale[i] <- scale[j];
      if( i = curr_index)
            increment i;
            new_scale[i] <- new_bound;
      fi
      increment i,j;
   end while
   free (scale);
end Split_scale
```

```
Merge_scale(scale, curr_index, no_of_boundary)

   allocate new_scale the size of which is decreased by 1;
   i,j <- 0;
   while ( i < no_of_boundary)
      if ( i is not equal to curr_index)
           new_scale[j] <- scale[i];
           increment j;
      fi
   end while
   free( scale);
end Merge_scale
```

We keep in primary memory only the most recently used directory and scales, and a data bucket. Therefore, if we are going to access a different directory bucket or data bucket, we should replace current ones with the new ones. Two global variables, curr_RBN and currDR_RBN, represent the currently active data bucket and directory bucket, respectively. The replacement procedure is included in the following search algorithm.

Algorithm 6

```
Find(grid_file,record)

   do for each resident scale
       curr_DR_index <- Index(resident_scale,key,.);
   od
   Access curr_directory bucket with curr_DR_index;
   if accessed directory bucket is not equal to
                                          currDR_RBN ,
      load new scale and directory from the bucket;
      currDR_RBN <- accessed RBN;
   fi
   do for each scale
       curr_index <- Index(scale,key,no_of_boundary);
   od
   Access current data bucket with curr_index;
   if accessed RBN is not equal to curr_RBN
       load new_bucket;
       curr_RBN <- accessed RBN;
   fi
```

```
    compare record with every existing record in current
    data bucket pointed by curr_RBN

    if same record is found
        return its position in the bucket
    else
        return(unsuccess)
    fi
end Find
```

Insertion algorithms

If insertions result in an overflow in the corresponding data bucket, splitting should occur. It is necessary to define the splitting of a region along a split boundary b_i of dimension D_i. Let the whole data space S be $D_0 \times D_1 \times \ldots \times D_{k-1}$ in k dimensional data space. If b_i is not included in D_i, the region of that dimension remains unchanged by splitting. Otherwise, let $D_i = [min_i, \ldots, max_i)$, the boundary values of which are kept in scale, s_i; splitting the region results in two new regions:

$$S_1 = D_0 \times \ldots \times D_i[min_i, \ldots, b_i) \times \ldots \times D_{k-1},$$
$$S_2 = D_0 \times \ldots \times D_i[b_i, \ldots, max_i) \times \ldots \times D_{k-1}.$$

We call region S_1 the lower region and region S_2 the upper region. A point data $(key_0, key_1, \ldots, key_{k-1})$ is said to locate at the lower region of b_i if $key_i < b_i$, and at the upper region of b_i otherwise. A data bucket is split along b_i by creating a new data bucket; then moving all the records located in the opposite region to the current region, where the record to be inserted locates, into the new bucket. If it happens that there is no movement of records due to data clustering, recursive splitting is needed. If the data bucket involved in

splitting is being pointed by a single block, directory splitting is triggered along the dimension which has largest region level. The procedure is described by the following algorithms.

Algorithm 7

```
Split(bucket, si , directory)

   if the data bucket is pointed by several directory
      blocks
        call split_bucket(curr_RBN,new_RBN) only;
   else
        call split_scale(si ,curr_index,no_of_boundary);
        call split_directory(DR,curr_index,Di );
   fi

end Split


Split_bucket(curr_RBN,new_RBN)

   call get_region(si ) to get bucket region;
                /*  get split boundary, bi   */
   bi <- (low_of_region + up_of_region) / 2;
   if record ri in current bucket <  bi
       do for all records rj in current bucket
           if rj is in upper region of bi
              move rj to the new bucket;
           fi
       od
   else
       do for all records rj in current bucket
           if rj is in lower region of bi
              move rj to the new bucket;
           fi
       od
   fi
   rearrange mapping of directory to data buckets in the
   whole bucket region;
end Split_bucket


Split_directory(DR,idx,Di )

   new_size <- n0 * n1 *...* (ni +1) *...* nk-1 ;
   allocate new directory of new_size;
   name all blocks below split index in Di ;
   copy  entry Di ni  at Di ni +1 ;
```

```
   rename all blocks above split index in Di ;
   free old directory;

end Split_directory

        cf. ni = number of intervals in si of Di


Insert(grid_file,record)

   call Find(record);
   if record does not exist
      if bucket is not full
         insert record into bucket;
      else
           call split(scale,directory,bucket);
           Insert(grid_file,record);
      fi
   else
       'duplicate record error';
  fi

end Insert
```

Deletion algorithms

As discussed in the previous chapter, continued deletions of records in a data bucket drops the bucket occupancy below a certain threshold. Then we need to merge the bucket to maintain a reasonable average bucket occupancy ratio. Bucket merge makes a bucket region in the directory by definition and the region level of the merge dimension decreases by one. Then there may be some boundary that no longer needs to be kept in directory when it is shared in the whole span of a dimension. In this case, we may merge directory and scale. We define the merge operation along a merge boundary m_j of dimension D_i as follows.

Let the whole data space S be S_1 such that:

$S_1 = D_0 \times \ldots \times D_i [min_i, \ldots, m_{j-1}, m_j, m_{j+1} \ldots, max_i) \times \ldots \times D_{k-1}$

The interval (m_{j-1}, m_j) and (m_j, m_{j+1}) in D_i is merged into one

interval (m_{j-1}, m_{j+1}) resulting S like S_2 such that:

$S_2 = D_0 \times \ldots \times D_i [min_i, \ldots, m_{j-1}, m_{j+1}, \ldots, max_i) \times \ldots \times D_{k-1}$.

Merge occurs at the same leaf level in view of binary split tree if we represent region level as in Figure-19. Deletion operations will be done by the following algorithms. As in insertion, file read/write operation is hidden in the algorithms.

Algorithm 8

```
Merge(bucket, scale,directory)
.
   call Candidate(cand_bucket,cand_dim);
   if there is no valid cand_bucket
        return(not mergeable);
   else
       call Merge_bucket(cand_bucket,cand_dim);
       if merge_bucket results in need of directory merge
          call Merge_directory(DR,curr_index,dim);
       fi
   fi .
end Merge


Candidate(cand_bucket,cand_dim)

   do for k dimensions
     get region buddy bucket;
     if (buddy_rec_cnt + to_be_merged_bk_cnt) is less than
        the upper_threshold AND their region level are
        same in  k dimensions then
       cand_bucket <- region buddy bucket;
     fi
   od

   if valid cand_bucket exist
      comparing region levels of sharing dimensions,
      select cand_bucket of largest region;
      cand_dim <- sharing dimension of largest region;
      return(cand_bucket, cand_dim);
   else
      return(no candidate);
   fi
end Candidate
```

```
Merge_bucket(cand_bucket,sharing_dim)

   i, j, k <- 0;
   while( i,j,k < bucket size)
       while(to_be_merged_bk[j] exists)
         j <- j + 1;
       end while
       while(cand_bucket[k] does not exist)
         k <- k + 1;
       end while
       if ( k >= bucket size)
           break;
       fi

       to_be_merged_bk[j] <- cand_bucket[k];
       increment i,j,k by one;
   end while
   rearrange mapping of directory to buckets;

end Merge_bucket


Merge_directory(DR,idx,Di )

   new_size <- n0 * n1 *...* (ni -1) *...* nk-1 ;
   allocate new directory of new_size;
   name all blocks below merge index idx in Di ;
   merge off blocks of idx in Di ;
   rename all blocks above idx;
   free old directory;
end Merge_directory


Delete(grid_file,record)

   call Find(record);
   if record exists in curr_RBN
       erase record from the bucket;
       decrement record count;
       check bucket occupancy;
       if the occupancy is less than low_threshold
           call Merge(bucket,scale,directory);
       fi
   else
        'record does not exist';
   fi

end Delete
```

CHAPTER VI

THE PROGRAM STRUCTURE

Grid file program

The grid file structure was programmed in C-language and run on Eunice operating system of VAX 11/780. We implemented the structure with processing two keys of long integer values generated by two independent random number generating programs. We built the grid file structure based on double level directory structure, which adopts resident directory scheme. For the simplicity, we separated the file structure into four files: resident scale file, resident directory file, directory bucket file, and data bucket file. However, we made it invisible to users. From a user's perspective, the grid file structure is a single file. The user interface operations discussed below are invoked by referencing a single file name. The overall structure of the grid file is shown in Figure-20. Our program consists of two parts: procedures for building the grid file structure and user interface procedures.

The file building procedures are performed with the algorithms discussed in Chapter V. The structure of our program described in program design language is included in Appendix D. Our program of grid file structure provides with

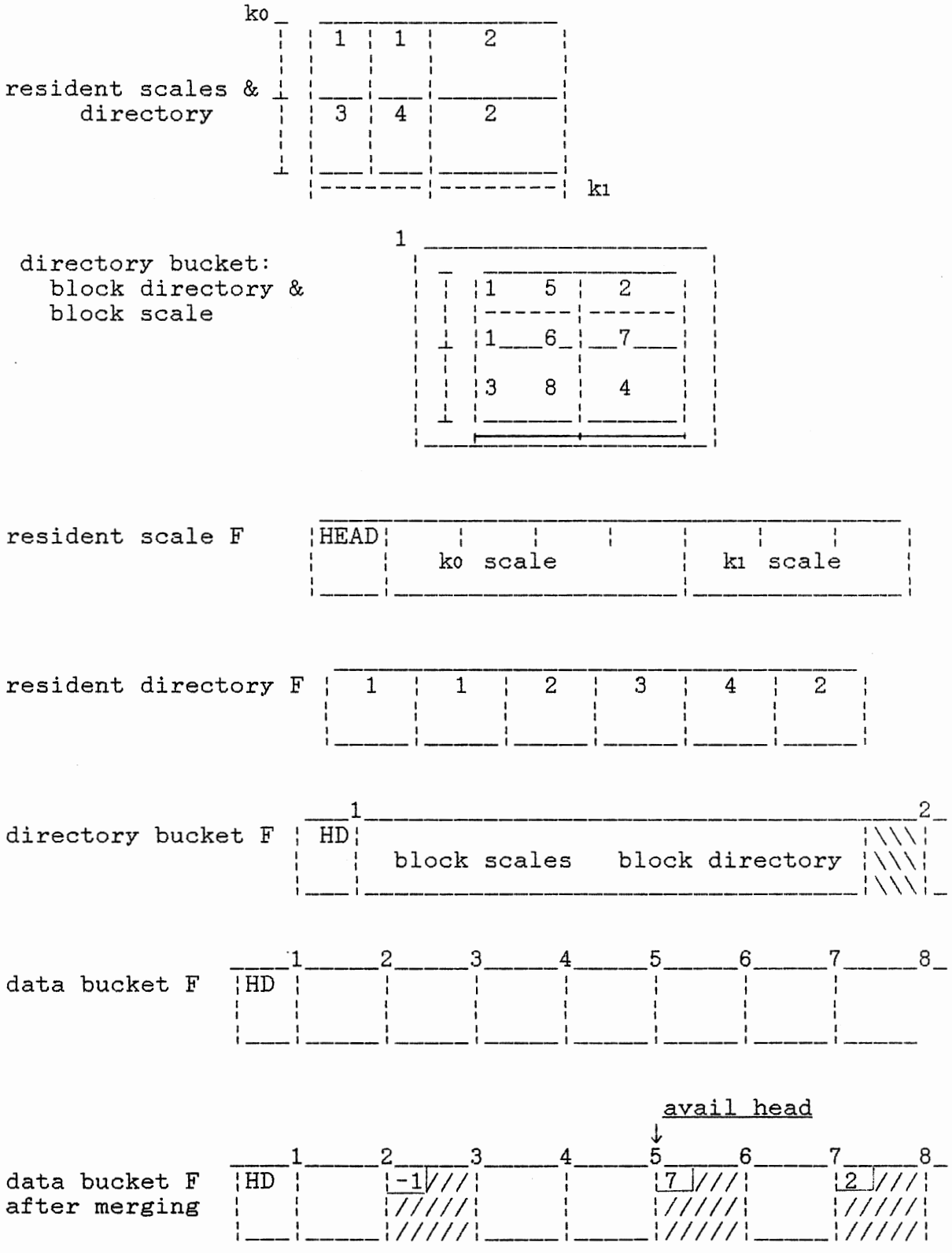


Figure-20. Structure of a grid file program.

the following operations:

- creating, opening and closing a grid file.
- inserting and deleting records in a grid file.
- updating a record in a grid file.
- searching a record in a grid file.
- range query.

The operations included in first and second groups may be performed with batch input files as well as interactively through terminals. Updating is a simple procedure that finds a specified record in a grid file and that if the record exist in the file, allows changing the associated information. The fourth and fifth groups are included in query operations. We discuss them in the following section.

Range query program

The grid file structure is a multikey searching structure of a data set that is characterized by a small number of attributes but the domain of each attribute is large and linearly ordered. Multikey searching structures allow a number of different searching types, each appropriate for answering a certain kind of query. The query type is usually classified into four categories: exact match query, partial match query, range query and best match query.

Exact match query is the simplest type among them. This searches for a specific record defined by the full attributes, k keys in k dimensional structures. We can use the Find() algorithm directly for this kind of query. The grid file

established aims to guarantee no more than two disk accesses for both successful and unsuccessful search in this query. A partial match query is a more complicated type of query in a multikey file that specifies a subset of attributes, t keys among k keys, t < k. In this query, we ask for all records that have those t values, independent of the other (k - t) attributes. This can be done in the following range query if we accept the whole range of each domain of the (k -t) keys.

In a range query we specify a range of values for each of the k keys, $[low_i, up_i]$, 1 <= i <= k, and all records that satisfy the range are reported. One of the design objectives of a grid file structure is to build an efficient structure for the range queries. For this purpose, a grid file partitions the whole data space into grid cells. Hence, we need not check all other attributes as well in obtaining a specified range in one attribute domain. That is, we do not need to do a recursive search as in k-d-tree implementation. It is very difficult to compare query efficiencies of different search structures since they are based on different design concepts. Saritepe[37] attempted an analytical comparison between grid file and k-d-B-tree structures. Though she considered grid scale accesses and directory accesses at the same level in node access count, which lessen the grid file performance, she showed that k-d-B-tree performance drops rapidly as the recursive partition frequency grows.

We give below a range query algorithm in a grid file. A constraint is imposed on the algorithm such that we must allow

only one disk access each per data bucket and directory bucket in searching for the records that satisfy the range specified in the range query. Though in the algorithm we check every record to see whether each key value satisfies the query region, we do not need to do so if the bucket region in the domain is completely contained in the query region. Since the internal structure of records in a data bucket is not of direct interest of grid file design, we insert records randomly into data buckets in our implementation. But we sort them easily within bucket boundary, showing at least partially sorted form for query reporting.

Algorithm 9

```
Range_query(grid_file)

  do for k dimensions
              /* get query range interactively */
     get lower_ and upper_bound for searching;
              /* call Index()   */
     get begin_resident_index with lower bound;
     get end_resident_index with upper bound;
  od

  allocate directory_bucket_queue;

  do for all resident directory blocks in the range
      get directory_bucket_RBN;
      if this RBN is not in directory_bucket_queue
          insert the RBN into the queue;
          load the directory_bucket pointed by the RBN;
      else
            break;
      fi

      do for k dimensions
       begin_index <-  Index
                          (scale,lower_bound,no_of_boundary);
       end_index <- Index(scale,upper_bound,no_of_boundary);
      od

      allocate data_bucket_queue
```

```
    do for each block traversing  directory bounded by
                  begin_index and end_index.
         get data_bucket_RBN;

         if this RBN is not in data_bucket_queue
            insert the RBN into the queue;
            load the data_bucket pointed by the RBN;
         else
               break;
         fi

        sort records in the data_bucket;
        report records within query range;
     od

     free ( data_bucket_queue);
  od

  free ( directory_bucket_queue);

end range_query
```

As shown implicitly in the range query algorithm, our program also uses a single fixed page replacement policy. Besides the resident directory and scales, the program keeps in primary memory only the most recently used block directory and its scales(not whole of directory bucket) and a data bucket. We assume that resident directory size could be small enough to be kept in primary memory during operations in most applications. We can show it as following simple case.

Let us define size parameters in our grid file structure using resident directory scheme as follows:

r = average size in bytes of an entry of resident directory used to point a directory bucket,

rs = size of the resident directory in bytes,

ps = size of a directory bucket in bytes,

d = average size in bytes of an entry of a block directory used to point a data bucket,

bs = size of a data bucket in bytes,

c = size of a record in bytes.

Then the grid file structure accommodates approximately the following data volume, v or number of records, n:

v = (rs/r) * (ps/d) * bs in bytes,

n = v / c.

Let ps and bs be 512 bytes. Implementation of our grid file structure showed about 69.6 percent data bucket occupancy. We expect a slightly lower percentage of directory bucket occupancy. This factor, of course, is not necessary in a single level grid file. If r and d are both 12 bytes, 10 k bytes of resident directory in our grid file is able to handle about 18.2 Mbytes. This means that the structure can also process about 284,500 records each 64 bytes long. Hence, a few ten kbyte of resident directory is sufficient for a practical application of grid file structure. In this regard, scale size can be negligible compared with that of directory. If we assume that every scale has b intervals, directory size has $O(b^k)$ and size of scales has O(kb). Since the entries of scales are boundary values of intervals, we can use a long integer for any kind and any length of key types converting them into canonical forms of long integer.

CHAPTER VII

CONCURRENCY CONTROL ON GRID FILE

General

A grid file is a large multikey access data structure. Of course, a single user may use the file system exclusively. However, we naturally expect that several different users access the data set simultaneously. As our structure is a multikey structure which is designed to process several attributes asymmetrically, we may think of it as a unification of several distinct data sets. Hence, it is more likely that many users share the data set manipulating it with a subset of the attributes of their own concern.

The file structure as well as the data set itself is not static as many users request a certain process to be executed on the structure such as insert, delete and find. When a large data structure is embedded on a system which allows a number of 'transactions' to be done concurrently, some kind of 'concurrency control mechanism' is needed as a part of the system to guarantee that concurrent transactions do not interfere with each other's operation. Efficient concurrency control has long been of interest especially in the database area. The objective of concurrency control is to prevent interference among users who attempt to access the shared data set simul-

taneously, keeping as high a level of concurrency as possible.

We acquire concurrency on a system by interleaving operations for different users. In the absence of proper concurrency control, the interference among the interleaved operations is apt to produce wrong results even though the operations are correct in themselves. We can see two typical cases of the problems as follows:

case 1: Lost updates.

Suppose two different branch offices of a bank attempt simultaneously to send their money to their headquarters on line. As illustrated in Figure-21, the two transactions may be interleaved resulting in a wrong balance at headquarters. We see that the update of Branch A was lost because the transaction of Branch B overwrites it.

time	HQ Balance	TR by Branch A	TR by Branch B
t1	$1,000,000.	-	-
t2	-	READ $1,000,000.	-
t3	-		READ $1,000,000.
t4	$1,700,000.	ADD $700,000.	-
t5	$1,200,000.	-	ADD $200,000.

Figure-21. Lost update.

Case 2: Inconsistent Information.

Suppose headquarters checks the total balance of the two branches and its own. If we assume that Branch A sends

$200,000 to Branch B, the information retrieved is wrong though there is no loss of update as shown in Figure-22.

time	HQ	Branch A	Branch B
t1	Balance $1,000,000.	$700,000.	$200,000.
t2	READ $1,000,000. b1 = $1,000,000.	-	-
t3	READ A $700,000. b1 = $1,700,000.	-	-
t4	-	SUB $200,000. b1_A = $500,000.	-
t5	-	-	ADD $200,000. b1_B = $400,000.
t6	READ B $400,000. b1 = $2,100,000.	-	-

Figure-22. Inconsistent information.

In a system in which concurrent access is allowed, users access the shared data expecting that the data satisfies certain consistency assertions specified for the system and that they get the result within a reasonable time. The basic problem in concurrency control is how to guarantee the correctness of a system state undergoing interleaved transactions. Eswaran et al.[12] established the notion of consistency and proposed the predicate lock method as a concurrency control mechanism. They suggested 'serializability' concept even though they did not use the term itself. Serializability[35,40] means that the effect of concurrent transactions should be the same as if the transactions have been run in a certain serial order. Rosenkrantz et al.[35] showed that serializability is both necessary and sufficient for consistency. It is generally used as the correctness

criterion of concurrency control mechanisms.

Many methods for achieving serializability have appeared in the literature. They can be classified into three categories such as locking, timestamping order and optimistic methods. First, in locking method[12,16] a transaction acquires a lock on a object so that it may not be changed in some unpredictable manner. There are generally two kinds of locks so called exclusive/shared lock or write/read lock. For the implementation of locking method, two protocols are needed such as granting and releasing protocol of lock(lock and unlock). Hence, basic locking method is called a two-phase-locking method(2PL). Unfortunately, a pure 2PL may lead to a deadlock situation in which a transaction waits indefinitely for its lock request to be granted. 2PL and its variants are most widely used in practical applications imposing some constraints to prevent deadlock. Second, in timestamping order method[33], every transaction is assigned a unique timestamp. A shared lock request for an object by a transaction is granted only if there is no other exclusive request with a larger timestamp. Similarly, an exclusive lock request on an object is accepted only if there is no other exclusive or shared lock on the object with a larger timestamp. Third, the optimistic method[19] is based on the assumption that conflicts among transactions in real application are quite unlikely. In this method, every transaction is allowed to perform its executions without any control. The history of a transaction is collected and a validation step is performed at the end of each

transaction to determine whether or not to commit the transaction.

Applications

We may separate large data structures into two classifications from several points of view such as static and dynamic structures, or tree and address computation structures, or single and multiple key structures. A concurrency control mechanism for static structures may be relatively simple and essentially is included in that for dynamic structures. In view of multiuser environment of large database area, it is natural that there be strong demand for efficient concurrency control mechanism. Since in current dynamic databases, tree structures have been widely used as indices, it is also natural that many researchers in the field of concurrency have been attracted to building the mechanism for the tree structures. Actually, a number of papers have appeared in the literature as solutions to concurrency problems in tree structures. They include for examples, [2], [18], and [22] for B-trees and its variants, and [25] for binary tree structures. In contrary, quite a few solutions have been published for hashing structures. We may refer to [8,9] for extendible hashing and [7] for linear hashing. Nobody proposed any solutions of concurrency problems for multikey structures including a grid file.

Most of the solutions are based on two-phase-locking with slight modifications of the data structures and imposing a

certain ordering in granting lock requests in order to prevent deadlocks. Locking methods remain still the most popular scheme in concurrency control mechanism. If we assume that locking method is also available for a (single level) grid file, we can suggest some design policies as follows under the general design objective that the mechanism allow a high degree of concurrency among user interface procedures.

- minimize the number of locks held at one time by a process.
- minimize exclusive lock and its time.
- make each process independent as much as possible.

We have three basic user interface procedures: FIND(), INSERT() and DELETE(). Insert and delete procedure may invoke SPLIT() and MERGE() respectively. We have also three basic entities which are to be locked: scale , directory and bucket. However, directory is defined by scales, we had better consider these two entities as an entity named as directory. First of all, we may consider split bucket and merge bucket operations independent with insert and delete operations though those are called by these procedures. The reason is that split bucket is triggered whenever the bucket overflows without being affected by other procedures and decision for bucket merge is made after delete procedure commits. Hence, we can defer the restructuring operations. During the operations, we create new buffers and build new ones based on the old versions to be restructured. Exclusive locks are to be requested at the end of the operations when we need change the pointers to the

entities. So, find operations may be done almost in parallel with any other operations. Insert and delete of data may operate in parallel if they are working on different bucket. We may take the inherent advantage of grid file such that access path to bucket are disjoint and the depth of access is shallow compared to those of tree structures.

Building a correct and efficient concurrent algorithm is not simple. In addition to the problems discussed so far, we have to consider the problems which may arise in multi-programming and parallel processing environments. Finally, we must take into account of the recovery problems together with concurrency control[1].

CHAPTER VIII

SUMMARY AND CONCLUSIONS

A grid file is a large multidimensional dynamic structure which uses address computation techniques . A single level grid file consists of three abstract data types, namely, linear scales, a directory and data buckets. A double level grid file maintains the directory in two levels such as a resident directory and block directories. Data buckets are fixed-sized structure units for storing data sets. A directory is used to manage data sets dynamically. Linear scales define the grids of a directory. Three major design objectives of a grid file are: (1) time bound of two disk accesses to search for a point data in disk memory; (2) reasonable average bucket occupancy; (3) efficient processing for range queries.

To realize the three design objectives, three basic design strategies are established: (1) maintaining a grid directory, elements of which are pointers to data buckets; (2) splitting with only two buckets involved; (3) grid partitioning the embedding space of a whole data set.

We have implemented a double level grid file of two dimensions. In a double level grid file, a directory is partitioned in grids and the corresponding sets of blocks of the directory are also stored in fixed-sized buckets, namely,

directory buckets. A resident directory is used to manage dynamically the growing and shrinking sets. Its entries are pointers to the directory buckets. A resident directory scheme is proposed to lessen disk accesses in neighborhood operations for processing geometric data sets which require that all dimensions be treated symmetrically[15]. In choosing a splitting dimension, we check region level in all dimensions and select the dimension which has smallest value. A split boundary is obtained by binary buddy system which for the value bisects the lower and upper boundary value of the interval to be split. In directory merging, we also adopt binary buddy system.

Our simulation studies are carried out with the following objectives:

I. Evaluation of memory utilization

(1) Bucket occupancy ratio
(2) Efficiency of resident directory
(3) Efficiency of directory

II. Estimation of processing time

(1) Insertion cost
(2) Deletion cost

Our simulation studies are done with key attributes of both long integer values obtained by two independent random generator programs. The following table shows the memory utilization statistics when we inserted 600 and 1,200 records.

record size	16	16
record count	600	1,200
size of data buckets in bytes	256	256
number of records per data bucket	16	16
size of directory buckets in bytes	512	512

number of blocks in resident directory	4	12
number of directory buckets	4	9
number of data buckets	56	109
average bucket occupancy	0.6690	0.6877
average directory bucket occupancy	0.5461	0.5280

average number of blocks in resident directory per directory bucket(r)	1.17
average number of blocks in block directories per data bucket(b)	2.5

During the insertions, we checked the intermediate statistics every 40 record insertions and computed averages of each value of the middle 4 lines in the above table. The following table shows the results:

grand average number of blocks in resident directory per directory bucket	1.25
grand average number of blocks in block directory per data bucket	2.1

Our studies show the same result in average data bucket occupancy as that of a single level grid file[27]. It shows about 0.69 occupancy ratio throughout the insertion period. Our grid file structure keeps the occupancy ratio at steady state where insertion and deletion frequencies are almost the same. It maintains over 50% occupancy until the data set size decreases by continuous deletions to 40% initial steady state. The grand average number of blocks in block directory per data bucket is a parameter to show the directory size in a single level grid file. Nievergelt et al.[27] show that b fluctuates around 2 as the number(record_count/bucket capacity) increases. With this regard, they recommend 10 or more for data bucket capacity(c) in applications. The parameter value(b)

in our studies show similar results as their's. Furthermore, we expect naturally the same result, r = 2, in the relation between a resident directory and directory bucket since both data bucket and directory bucket are split when the buckets overflow without any other conditions. The number, r = 1.25, in the above tables is because a resident directory has split only a few times and there is a little possibility to make bucket regions. The efficiency of directory shows the same level in both a single level and a double level grid file. It is because the data sets used for both simulations are obtained by random number generator programs. In practical applications of a grid file(c > 10) where data clustering is likely, more than two directory entries are needed for a data bucket and also more than two entries of resident directory are needed for a directory bucket. It is clear that the number becomes greater in a single level grid file because it split the whole span of the embedded data space while a double level confines split in a directory bucket. In this regard, The characteristics of data set suitable for a grid file are: (1) a small number of attributes(k < 10); (2) the domain of each attribute is large and linearly ordered; (3) attributes are independent each other. Since the size of directory entry is quite small compared to a bucket size, directory size causes no problems. We discussed the size of a resident directory and showed that it is small enough to be kept in primary memory. Considering the possible internal fragmentation(low occupancy ratio) in directory buckets though a double level grid file confines

directory split in a directory bucket, it is not yet known whether a single level grid file is more efficient or not in view of memory utilization.

Though a grid file has a time bound of two disk accesses for searching a point data, not much has been studied about time cost for insertion and deletion which are accompanied by splitting and merging. Each of a single insertion and a single deletion requires four disk accesses in a double level grid file(two accesses for searching and each one access for writing data buckets and directory buckets). Overall factors and times of disk accesses for a single insertion and a single deletion in a double level grid file are:

I. Insertion cost

Factors	Disk accesses
searching	2
data bucket split	2
directory split	1
directory bucket split	2
resident directory split	0
mapping directory to data bucket	0
total	7

II. Deletion cost

Factors	Disk accesses
searching:	2
data bucket merge	3
directory merge	1
directory bucket merge	3
resident directory merge	0
mapping directory to data bucket	0
total	9

The distribution of probabilities of each operations are not studied yet. It may depend on implementation strategies. If we are given an average values in the distribution of the probabilities, we can get average time cost in view of disk access. Splitting and merging may require many disk accesses especially in those operations on directory. A single level grid file which uses a conventional row major order array for a directory requires reorganizing of the entire directory when splitting and merging are needed. A number of disk access are needed in this kind of split and merge operations. A double level grid file minimize the disk accesses by confining the operations in a block directory and a resident directory which resides always in primary memory. So in view of time cost of insertions and deletions, a double level grid file is more efficient.

We remain some work to be done as further studies. It should include:

(1) Quantify the simulation studies,
(2) Implement other strategies and compare performance,
(3) Implement concurrency control on a grid file.

BIBLIOGRAPHY

(1) Agrawal, R., and Dewitt, D.J. "Integrated concurrency control and recovery mechanism:Design and performance evaluation." ACM Trans. Database Syst., 10, 4(Dec. 1985), 529-564.

(2) Bayer, R., and Schkolnick, M. "Concurrency of operations on B-trees." Acta Inf. 9(1977), 1-21.

(3) Bentley, J.L. "Multidimensional search trees used for associative searching." CACM 18, 9(Sept. 1975), 509-517.

(4) Bentley, J.L. "Multidimensional binary search trees in database applications." IEEE Trans. Softw. Eng. SE-5, 4(July 1979), 333-340.

(5) Bentley, J.L. and Friedman, J.H. "Data structures for range searching." Comput. Surv. 11, 4(Dec. 1979), 397-409.

(6) Burkhard, W.A. "Interpolation-based index maintenance." In Proc. ACM Symp. Principles of Database Systems(1983), 76-89.

(7) Chowdhury, S.K., and Srimani, P.K. "Worst case performance of weighted buddy systms." Acta Inf. 24, (1987), 555 -564.

(8) Cranston, B., and Thomas, R. "A simplified recombination scheme for the Fibonacci buddy system." CACM 18, 6(June 1975), 331-332.

(9) Ellis C.S. "Concurrence in linear hashing." ACM Trans. Database Systems, 12, 2(June 1987), 195-217.

(10) Ellis, C.S., "Distributed data structures:A case study." IEEE Trans. Comput. C-34, 12(Dec. 1985), 1178-1185.

(11) Ellis, C.S., "Extendible hashing for concurrent operations and distributed data." In Proc. ACM SIGACT-SIGMOD Symposium on Principles of Database Systems. ACM, New York, 1983, 106-116.

(12) Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L. "The notions of consistency and predicate locks in database system." CACM 19, 11(Nov. 1976), 624-633.

(13) Fagin, R., Nievergelt, J., Pippenger, N. and Strong, H.R. "Extendible hashing- a fast access method for dynamic files." ACM Trans. Database Syst. 4, 3(Sept. 1979), 315-344.

(14) Finsel, R.A., and Bentley, J.L. "Quad trees-a data structure for retrieval on composite keys." Acta Inf. 4(1974), 1-9.

(15) Hinrichs,K., and Nievergelt,J. "The grid file: a data structure designed to support proximity queries on spatial objects." In Proc. Workshop on Graph Theoretic Concepts in Computer Science, Osnabruck, (1983).

(16) Hsu, M., and Chan, A. "Partitioned two-phase locking." ACM Trans. Database Syst., 11, 4(Dec. 1986), 431-446.

(17) Kriegel, H.P. "Performance comparison of index structure for multikey retrieval." Proc. ACM SIGMOD, Boston, Massachusetts (1983), 186-196.

(18) Kwong, Y.S., and Wood, D. "A new method for concurrency in B-trees." IEEE Trans. Soft. Eng., 8, 3(May 1982), 211-221.

(19) Kung, H.Y., and Robinson, J.T. "On optimistic methods for concurrency control." ACM Trans. Database Syst. 6, 2(June 1981), 213-226.

(20) Larson, P.E. "Dynamic hashing." BIT 18(1978), 184-201.

(21) Lee, D.T. and Wong, C.K. "Worst-case analysis for region and partial searches in multidimensional binary search trees and balanced quad trees." Acta Inf. 9(1977), 23-29.

(22) Lehman, P. and Yao, S.B. "Efficient locking for concurrent operations on B-tree." ACM Trans. Database Syst. 6,4(Dec. 1981), 650-670.

(23) Litwin, W. "Linear hashing: a new tool for file and table addressing." IN Proc. 6th International Conf. on Very Large Data Bases, 1980, 212-223.

(24) Lloyd, E.L., and Loui, M.C. "On tne worst case performance of buddy systems." Acta Inf. 22, (1985), 451-473.

(25) Manber, U. and Ladner, R.E. "Concurrency control in a dynamic search structure." ACM Trans. Database Syst. 9, 3(Sept.1984), 439-455.

(26) Nievergelt, J. "Trees as data and file structures. "Lectures Notes on Computer Science, 112, (1981), 35 -45.

(27) Nievergelt, J., Hinterberger, H. and Sevcik, K.C. "The Grid File: An adaptable, symmetric multikey file

structure." ACM Trans.on Database Syst. 9, 1(Mar. 1984), 38-71.

(28) Orenstein, J.A. "Multidimensional Tries used for associative searching." Inf. Process.Lett. 14, 4(June 1982), 150-157.

(29) Otoo, E.J. and Merrett, T.H. "A storage scheme for extendible arrays." Computing, 31, 1(1983), 1-9.

(30) Oukel, M. and Scheuermann, P. "Multidimensional B-trees: analysis of dynamic behavior." BIT, 21, 4(1981), 401 -418.

(31) Overmars, M.H. and van Leeuwen, J. "Dynamic Multi dimensional Data Structures based on Quad- and k-d trees." Acta Inf. 17, (1982), 267-285.

(32) Peterson, J.L., and Norman T.A. "Buddy systems." CACM, 20, 6(June 1977), 421-431.

(33) Papadimitriou, C.H., and Kanellakis, P.C. "On concurrency control by multiple versions." ACM Trans. Database Syst., 9, 1(Mar. 1984), 89-99.

(34) Robinson,J.T. "The k-d-B-tree: a search structure for large multidimensional dynamic indexes." In Proc. SIGMOD Conf., ACM, New York, (1981), 10-18.

(35) Rosenkrantz, D.J., Stearns, R.E., and Lewis II, P.M. "Consistency and serializability in concurrent database systems." SIAM J. Comput. 13, 3(Aug. 1984), 508-530.

(36) Samet, H. "The quadtree and related hierarchical data structures." Computing Surveys, 16, 2(June, 1984),

187- 260.

(37) Saritepe, H.N.A. "An analytical comparison of grid file and k-d-B-tree structures." MS Thesis, Oklahoma State University, (Dec. 1987).

(38) Scheuermann, P., and Ouksel, M. "Multidimensional B-trees for associative searching in database systems." Inf. Syst. 7, 2(1982), 123-137.

(39) Tamminen, M. "The extendible cell method for closest point problems." BIT 22(1982), 27-41.

(40) Vidyasankar, K. "Generalized theory of serializability." Acta Inf. 24, (1985), 105-119.

APPENDIX A

ADDRESS COMPUTATION FOR DIRECTORY ACCESS

We have implemented a directory as a k-dimensional array. The directory is so large that we keep it in secondary memory. With the k index values obtained by INDEX() on k scales which define the directory, we can read a directory block corresponding to a bucket in one disk access. Given the address of the beginning of the directory, we can compute the relative address of a block.

```
directory: array[D0]....[Dk-1] of drelem.
drelem : a directory entry.
addr(x) : address of x.
sizeof(x) : size of x in byte.
```

The implementation of multidimensional arrays can be derived from that of one dimensional arrays. We see this for a two-dimensional array, and then generalize it to k dimensions.

In case of k = 2, D_0 = n and D_1 = m, we may consider directory [D_0][D_1] as an array A[1],...,A[n] in which each A[i] is in turn an array of m elements consisting the i-th row of the array(assume row major order). The address of A[i] is the sum of addr(A[1]) and the offset to i-th row. Then,

```
sizeof(a row of A) = m * sizeof(drelem)
addr(A[i]) = addr(A[1]) + (i - 1) * sizeof(a row of A)
           = addr(A[1]) + (i - 1) * m * sizeof(drelem).
```

The address of directory[i,j] is the sum of addr(A[i]) and the offset to j-th column. Therefore,

```
addr(directory[i,j]) = addr(A[i]) + (j - 1) * sizeof(drelem)
    = addr(A[1]) + [(i - 1) * m + (j - 1)] * sizeof(drelem)
    = addr(directory[1,1]) +
                  [(i - 1) * m + (j - 1)] * sizeof(drelem)
```

In a k-dimensional directory, we consider it as an array $A[a_1:b_1]...[a_k:b_k]$. Let sr = sizeof(a row of A) and se = sizeof(drelem).

When k = 1,

$$addr(A[i]) = addr(A[a_1]) + (i - a_1) * se. \qquad (1)$$

We may consider A to be an (k-1)-dimensional array, $A_{k-1}[a_1:b_1]....[a_{k-1}:b_{k-1}]$ of one dimensional array, $A[i_1,..,i_{k-1},j]$, $a_k <= j <= b_k$. Then, by (1),

$$addr(A[i_1,...,i_k]) = addr(A_{k-1}[i_1,...,i_{k-1}]) + (i_k - a_k) * se. \qquad (2)$$

Then,

$$addr(A_{k-1}[i_1,...,i_{k-1}]) = addr(A_{k-2}[i_1,...,i_{k-2}]) + (i_{k-1} - a_{k-1}) * sr$$

where sr = (bk - ak + 1) * se. Therefore,

$$addr(A[i_1,....,i_k]) = addr(A_{k-2}[i_1,...,i_{k-2}]) + (i_{k-1} - a_{k-1})(b_k - a_k + 1) * se + (i_k - a_k) * se.$$

If we apply (2) repeatedly, we get

$$addr(A[i_1,...,i_k]) = addr(A_1[i_1]) + se * \sum_{j=2}^{k} \left((i_j - a_j) \prod (b_m - a_m + 1) \right).$$

By (1),

$$addr(A_1[i_1]) = addr(A_1[a_1]) + (i_1 - a_1) * st,$$

where $addr(A_1[a_1])$ means the beginning address of the entire array, and st is the total size of an array of A_1.

st = se $\prod_{m=2}$(bm - am + 1). Finally, we get

$$addr(directory[i_1,...,i_k]) = addr(A[i_1,...,i_k]) = addr(directory[a_1,...,a_k]) + se * \sum_{j=1}^{k} \left((i_j - a_j) \prod (b_m - a_m + 1) \right). \qquad (3)$$

For k = 2, (3) shows as follow:

$$addr(directory[i_1,i_2]) = addr(directory[a_1,a_2]) + [(i_1 - a_1)(b_2 - a_2 + 1) + (i_2 - a_2)] * se.$$

If we substitute as follows:

$i_1 = i$, $i_2 = j$, $a_1 = a_2 = 1$, $b_2 - a_2 + 1 = m$, we get

addr(directory[i,j]) = addr(directory[1,1])
+ [(i -1) * m + (j - 1)] * se.

This is exactly the same as we have derived directly at the beginning. In the C-implementation of array arithmetic, indexes start with zero, se is a unit of a array, and array name has a address value. So, the address computation function is simplified as follow:

addr : address of a block with index values, i and j. Then,

addr = directory + i * b + j,

when we declare a directory as directory[a][b] of drelem.
For k = 3, we can derive the following formula following the procedures used above.

addr = directory + b * c * i + c * j + k,

when we declare a directory as directory[a][b][c] and the index values in each dimension are i,j,k respectively.

APPENDIX B

A SIMPLE EXAMPLE OF GRID FILE OPERATIONS

A simple example of insertion and deletion accompanied by splitting and merging operations is presented in this appendix.

k_0 : age, boundary value : 16 - 76.
k_1 : salary, " " : 8 - 128, unit:$1000.
k_2 : department no. " " : 000 - 800.

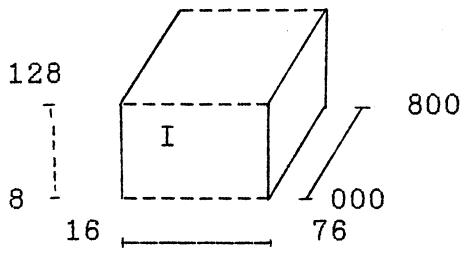
data bucket capacity = 2 records.
upper threshold for merging data bucket = 1 record.
directory bucket capacity = 4 blocks of a directory.
upper threshold for merging directory bucket = 2 blocks.
lower threshold for merging directory bucket = 1 block.

sequence of arriving of data record:

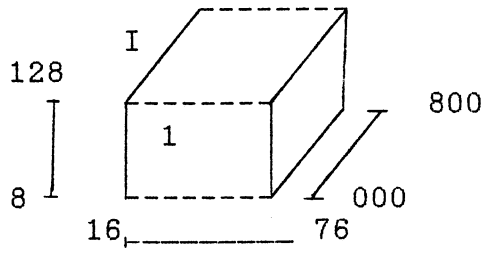
	name	k_0	k_1	k_2
insert	a	29	35	110
	b	70	110	200
	c	34	50	310
	d	60	92	700
	e	47	40	210
	f	45	70	510
	g	50	75	150
	h	55	73	450
	i	75	120	250
	j	65	35	750

delete b, g, i, h, e, a, j, c.

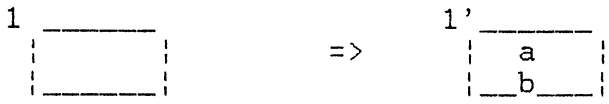
step 1. Initial state and a data bucket full.



resident directory

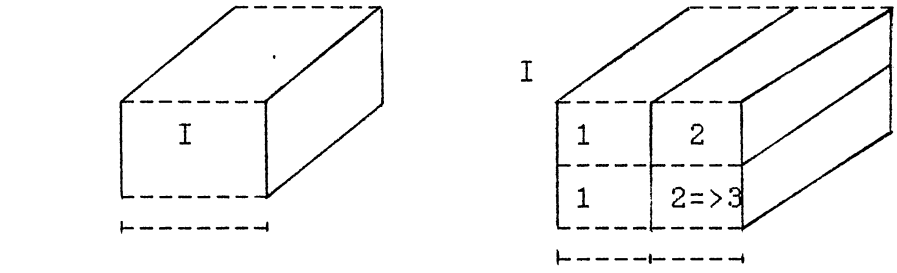


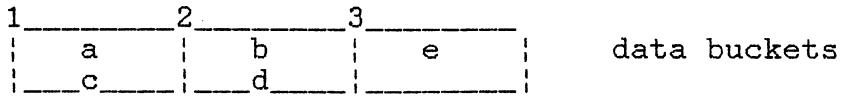
block directory



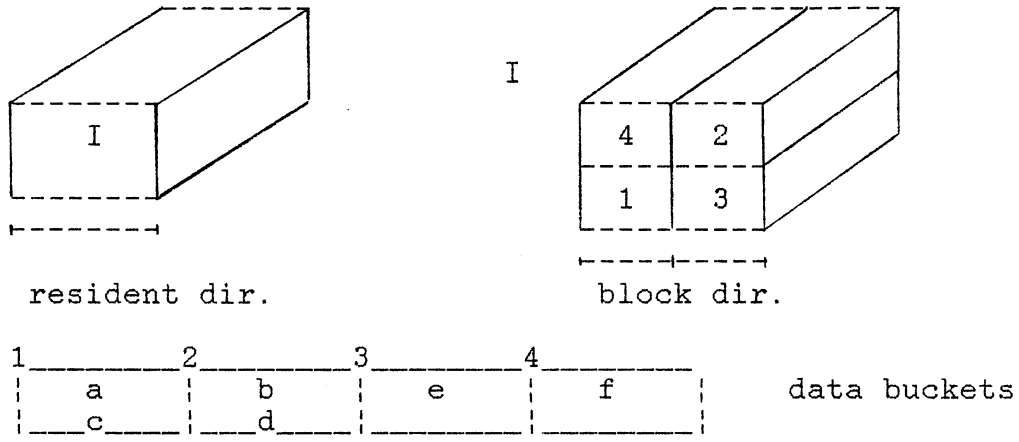
step 2. When record c arrives, find data bucket full.
Split directory along first dimension.
Split data bucket.
Insert c and d.

step 3. When record e arrives, find data bucket(#2) full.
Split directory along second dimension.
Split the data bucket and insert e.

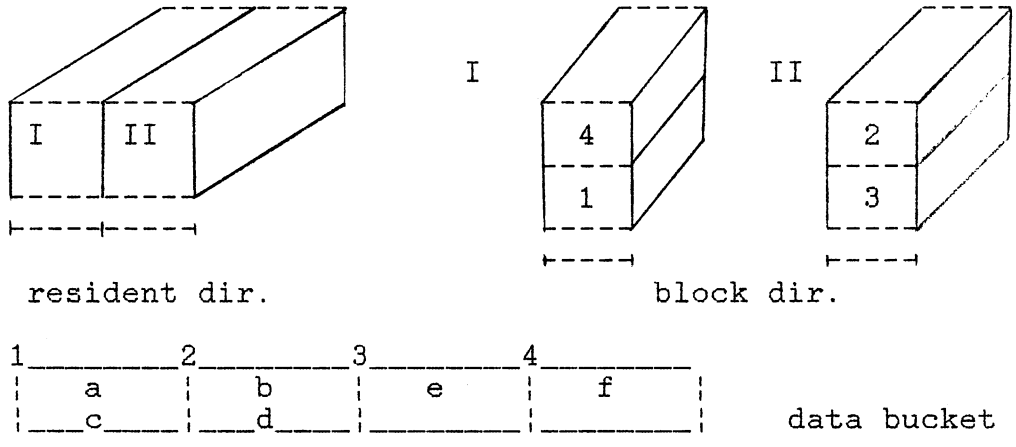




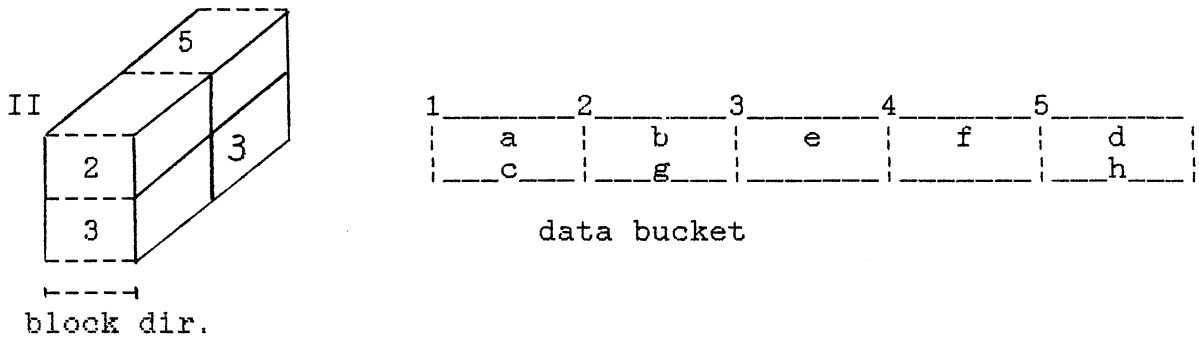
step 4. When record f arrives, find data bucket(#1) full.
Split data bucket. Insert f and adjust mapping.



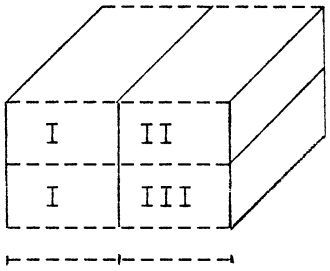
step 5. When record g arrives, find data bucket(#2) full.
Split resident directory along first dimension.
Split directory bucket along the split boundary of resident directory.



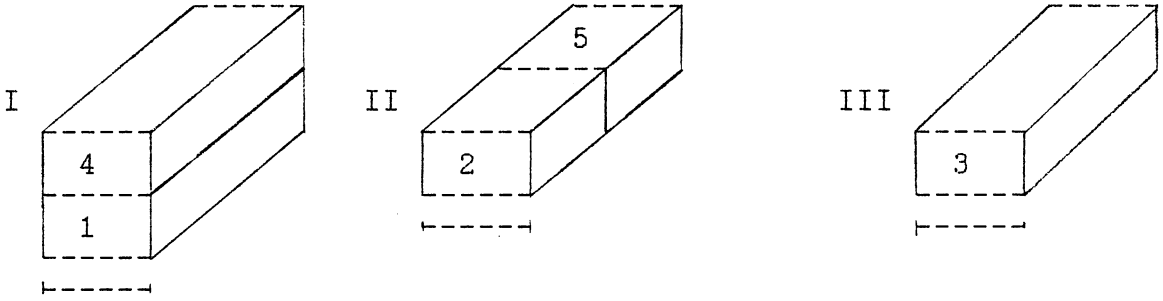
step 6. Split directory in dir bucket II.
Split data bucket(#2) and insert record g.
Insert h.



step 7. When record i arrives, find data bucket full.
Split resident directory along second dimension.
Split directory bucket along the split boundary of resident directory.

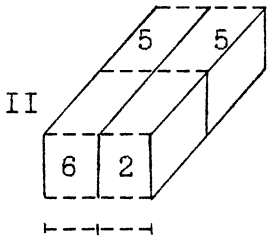


resident directory.

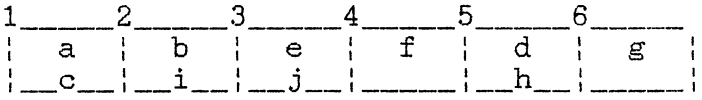


block directory.

step 8. Split block directory II. Split data bucket(#2).
Insert record i and j.

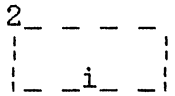


block dir.

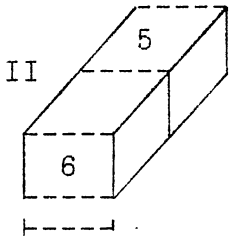


data buckets

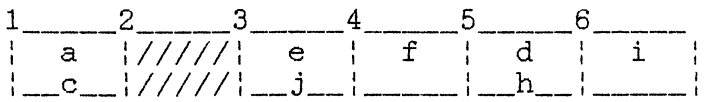
step 9. Delete b.



step 10. Delete g. Underflow occurred in data bucket(#6).
Find candidate #2 and merge it to #6(current bucket).
Merge directory II.

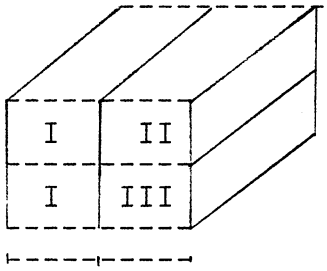


block dir.

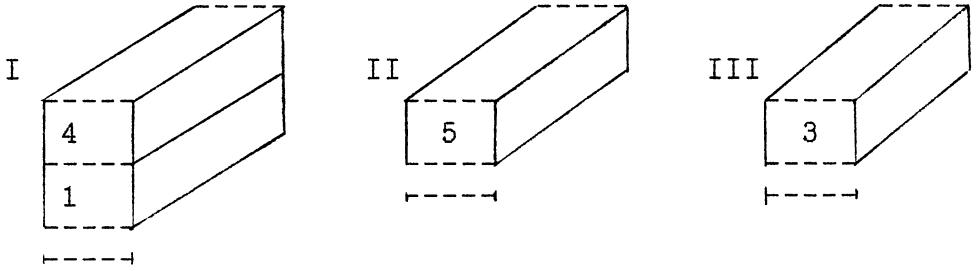


data buckets. //// : in avail list.

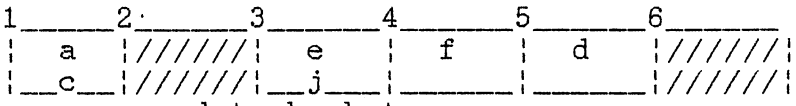
step 11. Delete i. No merge candidate.
step 12. Delete h. Underflow occurred in bucket #5 and found candidate bucket #6. Merge #6 to #5.
Merge block directory II.



resident directory.

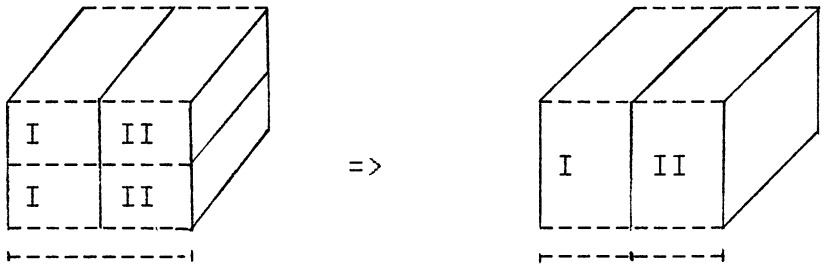


block directory.

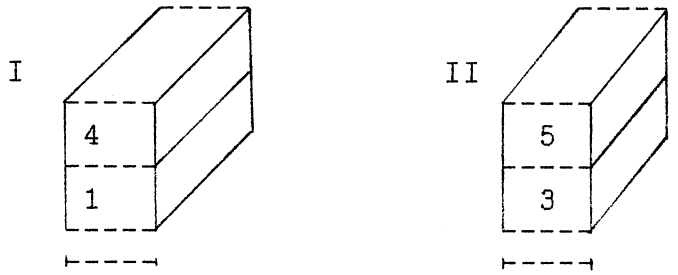


data bucket

step 13. Underflow in directory bucket II. Found merge candidate directory bucket III. Merge them and adjust mapping in resident directory.

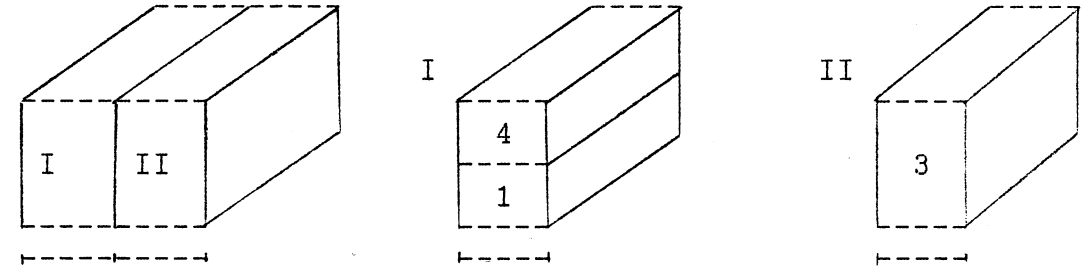


resident directory.

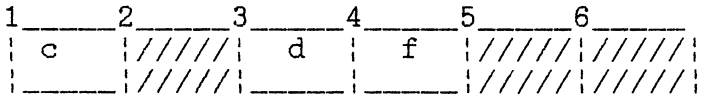


block directory.

step 14. Delete e, a and j. Underflow in data bucket #3. Find merge candidate #5. Merge block directory. Underflow in directory bucket II, but no candidate directory bucket.

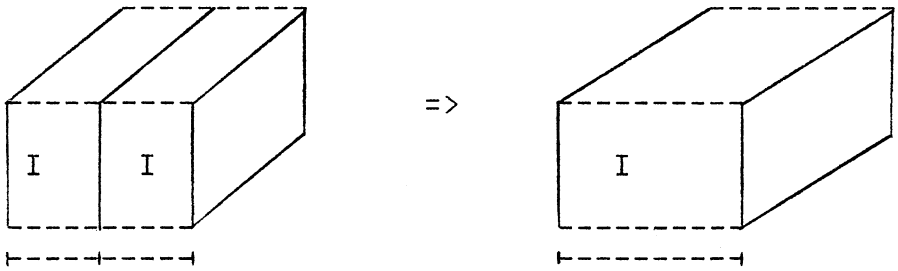


resident dir. block directory.

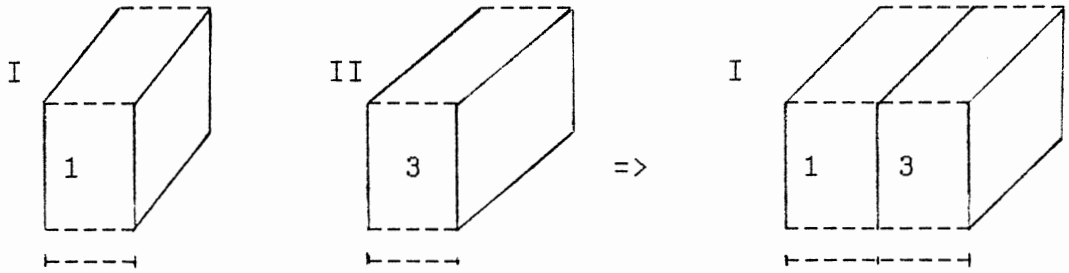


data bucket

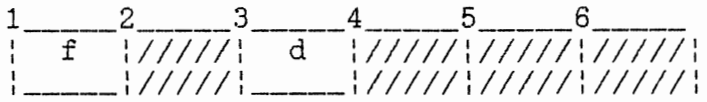
step 15. Delete c. Underflow in data bucket #1. Find merge candidate bucket #4. Merge them into #1. Underflow in directory bucket I. Find candidate bucket II. Merge them and adjust mapping in resident directory. Merge resident directory.



resident directory.



block directory



data bucket. ` ///// bucket in avail-list.

APPENDIX C

A DISCUSSION ON THE STRATEGIES OF CHOOSING SPLIT AND MERGE DIMENSIONS AND BOUNDARY VALUES

We discuss here the strategies of choosing a dimension to be split and a boundary value(split position) to be inserted in the dimension in some informal way. In choosing the split dimension, if we are given the characteristics of data set such as its distribution and some necessity to refine in a certain dimension, we can adapt our policy to that characteristics. Otherwise, we must establish a determinable sequence.

The simplest is a 'cyclic' method which selects one dimension in turn among all dimensions. This method works well in a single level grid file[27]. However, we can expect undesirable state in a double level grid file as discussed below.

A grid file is a multikey access structure which treats all keys symmetrically. Without any specific information of attributes of each dimension, we do not need to make them more refined in some dimension. We keep the refined-level as uniform as possible in all dimensions. The local levels discussed in main chapters define the refined-level in each dimension. We compute the levels with procedures on each

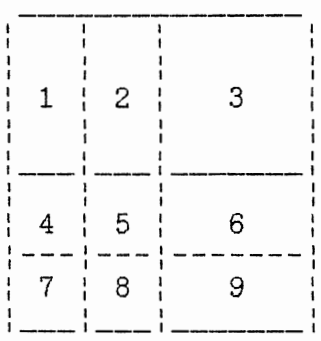
scale. Our algorithm for choosing a dimension to be split is as follow:

```
get_split_dim:

          /* subscripts denote dimensions */

     do for each dimension i  /* 1 <= i <= k    */
        levelᵢ <- get_level();  /*  by Algorithm 1 */
     od

     find the smallest levelᵢ ;
     if found  several dimensions of the same smallest value
        then
          numberⱼ <-  numbers of entries in a scale defining
                      the dimensions;
          find the smallest numberⱼ ;
          if found several dimensions of same numberⱼ then
            return(1);/*first dimension to get a unique dim */
          fi
          return(j);
     fi
     return(i);
end get_split_dim
```

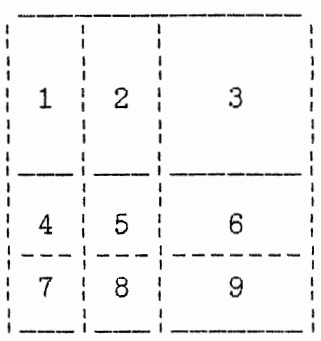
We may meet the same state of directory like State 1 in both the cycle method and level method(we call the method of the algorithm above hereafter) when we start splitting the directory at vertical dimension. The numbers in blocks of the directory are pointers to data buckets. Assuming that data bucket #1 is to be split and followed by directory splitting, each state is as State 2.

Comparing the two states, we see that local levels vary from one to three in the state by cycle method and from one to two in the state by level method. Using the same data set, we get different refinement among dimensions. We expect more fluctuation in query reponse time when levels spread more widely. In view of the neighborhood property, we prefer to the

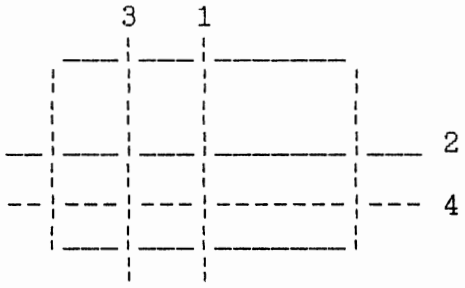
state 1:



By cycle method

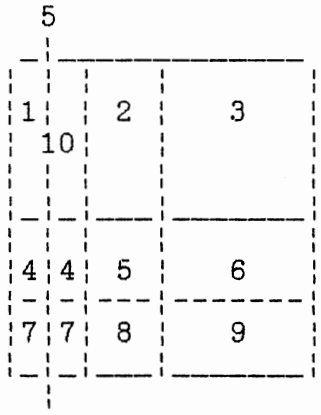


By level method

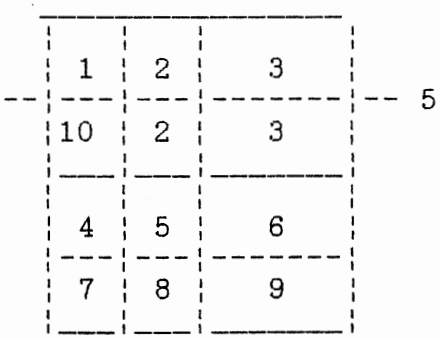


split sequence in both cases.

state 2:



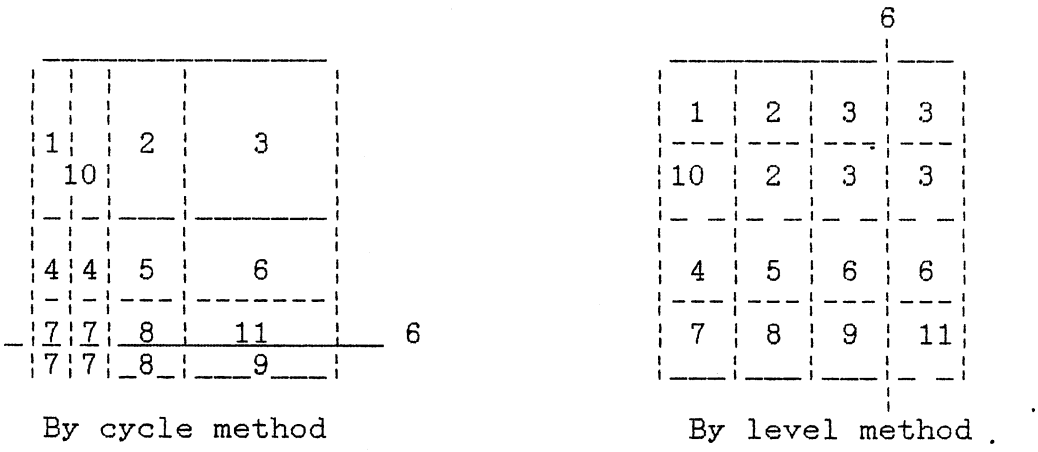
By cycle method



By level method

state by level method. Moreover, the simplicity of cycle method in choosing the dimension to be split can not be obtained in a double level grid file. We have to maintain the cycle sequence separately in grid directory and each block directory. Otherwise, the variance of level value becomes greater. We may meet state 3 when data bucket #9 is split and directory splitting follows in State 2.

state 3:



Assume that we split the directory bucket containing the block directory of state 3 along vertical dimension. We choose the split boundary which separate blocks corresponding to data bucket #2 and #3 in each block directory to keep the binary split constraints. In case of cycle method, one of new directory buckets contains four blocks and the other contains 12 blocks. In contrast, each new directory bucket has 8 blocks in level method. This implies that level method makes more 'adaptable' structure which is one of characteristics of grid

file. We expect more uniform response time in user interface insertions and deletions.

In choosing a boundary value for splitting and merging operations in grid file, we can adopt directly the splitting and merging policies of buddy systems on which recently many papers have appeared in literature[7,24,32]. These discribe the dynamic memory allocation mechanisms and their performance from the stand point of operating systems. There are three standard buddy systems: binary, Fibonacci and weighted buddy systems. We digress from our discussion to summarize their work. The average internal fragmentation of the binary buddy system is larger than that of the Fibonacci buddy system which is larger than that of weighted buddy system. The external fragmentation of binary buddy system is less than that of Fibonacci buddy system which is less than that of weighted buddy system. The total fragmentation of the three buddy system are almost the same showing around thirty percent value. With systems, they conclude there is a reasonable assurance that no better buddy system can be chosen without knowledge of the actual memory request distribution.

In a grid file, we split a data bucket into two and maintain a correspondence between the bucket and its embedded space location in a directory. In maintaining the directory dynamically, we can adopt the address computation methods of buddy systems for splitting and merging operations.

In binary buddy system, the entire memory space consists of 2^m words, which means the address space is $[0,2^m)$. This

method splits its memory spaces by bisecting them. So the address of a block of size 2^k is a multiple of 2^k. Memory sizes allowed are 1, 2, 4, 8, 16, ... In Fibonacci buddy system, the initial entire address space is size Fn and blocks size F_i are split into blocks of size F_{i-1} and F_{i-2} (F_i is the i-th Fibonacci number). So the allowed memory sizes are 1, 2, 3, 5, 8, 13,... In a weighted buddy system blocks may be of sizes 2^k, $0 <= k <= m$, and $3*2^k$, $0 <= k <= m-2$ when initial total memory size is 2^m. So the allowed block sizes are 1, 2, 3, 4, 6, 8, 12,...

In binary buddy system, we can easily compute the address of a buddy of a block given the block's address and its size as follows:

```
A : a block of size 2^k,
addr(A) : address of A.

if addr(A) mod 2^(k+1) = 0 then
     addr(buddy(A)) = addr(A) + 2^k;
else if addr(A) mod 2^(k+1) = 2^k then
     addr(buddy(A)) = addr(A) - 2^k;
fi
```

This address computation method is applied to our buddy system algorithms shown in Chapter V. The address computation in binary and weighted buddy systems are somewhat straightforward. But that of Fibonacci buddy system was not efficient until Cranston et al.[8] designed the method. We expect more complexity in choosing split boundary when a directory bucket is split. We can not find any reason to choose other than binary buddy system since the bucket occupancy and directory size are not to be affected by any of these three methods.

APPENDIX D

PDL DESCRIPTION OF A GRID FILE PROGRAM

The grid file program we have implemented for k = 2 dimensions is described in this appendix using program design language(PDL). We believe that this can be easily extended to higher dimension cases with slight modification. We have sometimes followed C-like statements in the descriptions. We denote variables with upper case letters in procedures. The global variables are explained at the beginning. But we have not declared every local variable in every procedure trying to make them self-explainable with their pseudo names. Indices of arrays begin with zero value. In most cases, ARRAY[0] is related with the first dimension and ARRAY[1] with the second dimension.

The global variables declared:

File descriptors and pointers:

```
rs_scfd - resident scale file.
rs_drfd - resident directory file.
drfd    - directory bucket file.
bkfd    - data bucket file.
datafp  - batch input data file.
outfp   - output file.
```

File pointers acting in primary memory:

```
rescale[k] - array of pointers to resident scales of k
             dimensions.
res_drf    - resident directory.
scale[k]   - array of pointers to block scales of k dimensions.
directory  - block directory.
```

Structured variables:

```
structure {  short  level[k],  - region level.
             short  shared; - show sharing dimension.
             short  cnt;   - record count.
             long   RBN;   - pointer to bucket.
          } drelem;        - entry of each directory.
structure {  long key[k]   - key values.
             char info[INFOLEN]  - non key information.
          } record, bucket[NO_REC];

structure {  long  bkcnt;
             long  availhead;
             long availcnt;
          } bkhead, drbkhead; - head of each bucket file.
```

Flags and Others:

```
reschead[k] - array of entry numbers of resident scale.
rs_idx[k]    - array of current index in each resident scale.
schead[k]   - array entry numbers of block scale.
idx[k]      - array of current index in each block scale.
rsaddr      - pointer to current cell in resident directory.
addr        - pointer to current cell in block directory.
curr_RBN    - pointer to currently acting data bucket.
currDR_RBN  - pointer to currently acting directory bucket.
*_changed - show if *_file is modified or not.
*_init    _ show if *_file is for created one or not.
```

Major defined variables:

```
NOSHARE = -1 : This means that current cell in directory is not
               a bucket region, e.g. a single block.
XSHARE   = 0  : This means that current cell forms a bucket
               regin with its neighbor(s) to the direction of
               first dimension.
YSHARE   = 1  : This means that current cell forms a bucket
               region with its neighbor(s) to the  direction of
               second dimension.
XYSHARE = 2   : This means that current cell forms a bucket
               region with its neighbor(s) to both directions
               of first and second dimensions.
```

These values are to be kept in the field 'shared' of the structured variable 'drelem' which is an entry of directory. The values are used extensively to distinguish the cases of merging and splitting discussed in Chapter IV. We have some more defined-variables that are explainable by themselves.

```
main:proc(argument);
        /* get arguments from O/S  */
  if argument number < 2 then
      message and exit; fi
  initialize flags;
  clear BUCKET with -1 in each key;

  if argument number is two then
     open files and assign file descriptors;
     load their headers calling load*(fd,case);
  else if argument number is three then
     creat files and assign file descriptors;
     initialize their headers calling load*(fd,case);
     initialize *_INIT flags with TRUE;
  fi

  if data file opened then  /* batch operation  */
     call build_gridfile(datafp); fi

  if DO_INIT = TRUE then  /*  for just created grid file */
     call statitstics(gridfile); fi
                     /*  user interface procedure  */
  call menu();
  do while(opcode < NO_FUNCT)
     when(opcode)
       0: get keys and call find(gridfile,RECORD);
       1: get RECORD and call insert(gridfile,RECORD);
       2: get keys and call delete(gridfile,RECORD);
       3: call update(gridfile);
       4: call range_query(gridfile);
     call menu();
  od
  if DO_INIT or RSDR_CHANGED then
        /*  gridfile is created or resident directory changed */
    write RESCHEAD[k] and RESCALE[K] to file RS_SCFD;
    write RES_DRF to RS_DRFD;
  fi

  if DR_CHANGED then  /* directory is changed */
     seek the beginning of DRFD;
     write DRBKHEAD;
     seek the position of current resident directory bucket.
     write SCHEAD[K], SCALE[K], and DIRECTORY to file DRFD;
  fi

  if BK_CHANGED then   /* bucket is modified */
     seek the beginning of file BKFD;
     write BKHEAD;
     seek the position of current data bucket;
     write BUCKET;
  fi
  close files;
end main
```

The following procedures are to load files into primary memory.

```
 loadrescale(): this is to load resident scale file from RS_SCFD.
 loadresdrf():  This is to load resident directory from RS_DRFD.
 loaddrf() : This is to load current directory from DRFD.
 loadbkf() : This is to load current data bucket from BKFD.

loadrescale:proc(fd,case)
       /* This is to load resident scale file */
   if CASE is for existing gridfile then
       seek the beginning of file RS_SCFD;
       read RESCHEAD[k];
       allocate RESCALE[k] with each size of RESCHEAD[k];
       read RESCALE[k];
    else     /* for a new created grid file  */
       initialize RESCHEAD[k] with each 2;
       allocate RESCALE[k] WITH each size of 2;
       initialize RESCALE[k] with MINBOUND[k] and MAXBOUND[K];
    fi
end loadrescale

loadresdrf:proc(fd,case)
      /* This is to load resident directory file */
   allocate RES_DRF with size of multiply of RESCHEAD[k]-1;
   if CASE is for a created gridfile then
       initialize entries of RES_DRF with 0 and NOSHARE each;
   else   /*  for existing gridfile  */
       seek the beginning of RS_DRFD;
       read RES_DRF from the file RS_DRFD;
       currDR_RBN =  -1 /*  means that current directory not
                  loaded yet */
   fi
 end loadresdrf

 loaddrf:proc(fd,case)
       /* This is to load current directory */
   if CASE is for a created grid file then
       initialize DRBKHEAD and SCHEAD[k];
       allocate SCALE[k] and DIRECTORY with initial sizes;
       initialize SCALE[k] and DIRECTORY with init values;
       curr_RBN = 0  /*  point to first data bucket  */
       write DRBKHEAD to DRFD;
   else if CASE is for a opened gridfile then
       read DRBKHEAD from the opened file;
       curr_RBN = -1;  /* There is no current data bucket */
   else  /*  replacement of directory bucket  */
       if DR_CHANGED then   /* directory was modified */
           seek currDR_RBN position of fd;
           write SCHEAD[k], SCALE[k] and DIRECTORY;
       fi
       seek new position for new directory;
       read new SCHEAD[K];
       allocate SCALE[k], DIRECTORY with SCHEAD[k] values;
```

```
        read SCALE[k], DIRECTORY from fd;
        DR_CHANGED = FALSE;
    fi
end loaddrf


loadbkf:proc(fd,case)  /* case shows a new bucket position */
    if CASE is for a created gridfile then
        initialize BKHEAD with inti values;
        write BKHEADinto file BKFD;
        curr_RBN = 0 /*  point first databucket  */
    else if CASE is for a opened gridfile then
        read BKHEAD from file BKFD;
    else  /*  replacement of current BUCKET  */
        if current bucket was changed then
            seek current data bucket position in BKFD;
            write current BUCKET;
        fi
        clear current BUCKET;
        seek and read BUCKET from the new bucket position in BKFD;
        BK_CHANGED = FALSE;
    fi
end loadbkf

build_gridfile: proc(datafp)
    do while get_record(datafp) not EOF
        if RECORD.KEY[0] < 0 then
              negate RECORD.KEY[0];
              call delete(BKFD,RECORD);
        else
              call insert(BKFD,RECORD);
        fi
    od
    close data file;
end build_gridfile

insert: proc(fd,RECORD)
    if record keys are not valid then
         message and return; fi
    if find(fd,RECORD) < 0 then    /* find() is a function  */
        if ADDR-> CNT < MAX_NO_REC then  /* bucket not  full */
            insert RECORD in current  BUCKET;
            increment ADDR->CNT;
            arrange mapping of directory;
            DR_CHANGED = BK_CHANGED = TRUE;
        else   /*  bucket full  */
              call split(gridfile);
              call insert(fd,RECORD);
        fi
    else
         message 'record found'
    fi
end insert
```

```
delete: proc(fd,RECORD)
   if record keys are not valid then
        message and return; fi

   if (i = find(fd,RECORD)) >= 0 then
             /*   i stands for the record position in bucket */
        clear record BUCKET[i];
        decrement  ADDR->CNT;
        arrange mapping of directory;
        DR_CHANGED = BK_CHANGED = TRUE;
        if ADDR->CNT < LOW_THRESHOLD then
             call merge(gridfile); fi
    else
         message 'record not exist';
    fi
end delete

find:function(fd,RECORD)
  do for all k
     RS_IDX[k] = index(RESCALE[k],RECORD.KEY[k],RESCHEAD[k]);
  od
       /* following computation is for the case: k = 2  */
  RSADDR = RES_DRF + RS_IDX[1] * (RESCHEAD[0]-1) + RS_IDX[0];
  if RSADDR->RBN != currDR_RBN then
      POS = new directory bucket position ;
      call loaddrf(DRFD,POS);  /* replacement of directory */
      currDR_RBN = RSADDR->RBN;
  fi

  do for all k
     IDX[k] = index(SCALE[k],RECORD.KEY[k],SCHEAD[k]);
  od
      /* following computation is for the case: k = 2 */
  ADDR = DIRECTORY + IDX[1] * (SCHEAD[0] - 1) + IDX[0];
  if ADDR->RBN  !=  curr_RBN then
      POS = new data bucket position;
      call loadbkf(BKFD,POS);  /* replacement of bucket */
      curr_RBN = ADDR->RBN;
  fi

  search RECORD in current BUCKET;
  if found then return the record position in BUCKET;
  else  return (-1);
  fi
end find

split: proc()
  if ADDR->SHARED > NOSHARED then/* curr dir is a bucket region*/
     call split_bucket(new_RBN,ADDR->SHARED);
  else
     DIM = get_split_dimension();
     if current directory will be overflowed then
        call split_resident();
     else
```

```
        call split_scale(SCALE[dim],IDX[dim],SCHEAD[dim],0);
        increment  SCHEAD[DIM];
        call split_dr(DIRECTORY,IDX[k],DIM,ISRES= 0);
        DR_CHANGED = TRUE;
    fi
  fi
end split

split_resident:proc
  if RSADDR->SHARED > NOSHARE then/* directory bucket region */
     split_dr_bucket(DRFD,RSADDR->SHARED);
  else
      DIM = get_split_dim();
      call split_scale(RESCALE[dim],RE_IDX[dim],RESCHEAD[dim],1);
      increment SCHEAD[DIM];
      call split_dr(RES_DRF,RE_IDX[k],DIM,isres = 1);
      RSDR_CHANGED = TRUE;
   fi
end split_resident


split_scale:proc(scl[i], INDEX[i], no_of_boundary,ISRES)
     get LOW_ and UP_BOUND of interval IDX in the SCL[i];
     NEW_BOUND = (LOW_BOUND + UP_BOUND ) / 2;
     allocate NEW_SCALE  the size of which is increased by 1;
     l, j = 0;
     do while  l < no_of_boundary
         l-th entry of NEW_SCALE = j-th entry of SCL[i];
         if l = INDEX[i] then
             increment l;
             l-th entry of NEW_SCALE = NEW_BOUND;
         fi
         increment l,j;
     od
     free SCL[i];
     if ISRES = 0 then    /* this is for block scale  */
        SCALE[i] = NEW_SCALE;
     else                 /* this ia for resident scale */
        RESCALE[i] = NEW_SCALE;
     fi
end split_scale


merge_scale:proc(scl[i], index[i], no_of_boundary,isres)
     allocate NEW_SCALE the size of which is decreased by 1;
     l,j = 0;
     do while  l < NO_OF_BOUNDARY
         if l  !=  INDEX[i] then
           j-th entry of NEW_SCALE = l-th entry of SCL[i];
           increment j;
         fi
         increment l;
     od
     free SCL;
```

```
     if ISRES = 0 then      /*  for block scale  */
        SCALE[i] = NEW_SCALE;
     else
        RESCALE[i] = NEW_SCALE;
 end merge_scale


split_dr:proc(curr_dr,idx[k],dim,sch[k],isres)

    NEWSIZE = (SCH[0] - 1) * (SCH[1] - 1);
    allocate NEW_DR with size of NEWSIZE;
    if DIM = 0 then
       CURR_SIZE = (SCH[0] - 2) * (SCH[1] - 1);
       i,j = 0;
           /* traverse current DIR to copy entries to new DIR */
       do while i < CURR_SIZE
          if i mod (SCH[0] - 2)) = idx[0] then
             if current SHARED < XSHARE  then
                current SHARED = XSHARE;
             else if current SHARED = YSHARE then
                    current SHARED = XYSHARE;
             fi
             j-th entry of NEW_DR = i-th entry of CURR_DR;
             j = j + 1;
          fi
          j-th entry of NEW_DR = i-th entry of CURR_DR;
          i = i + 1;
       od
    else if DIM = 1 then
       CURR_SIZE = (SCH[0] - 1) * (SCH[1] - 2);
       i,j = 0;
       do while i < CURR_SIZE
        if i < (SCH[0] - 1) * IDX[1] then
            i-th entry of NEW_DR = i-th entry of CURR_DR;
            j = j + 1;
        else if (SCH[0]-1) * IDX[1] <= i < (SCH[0]-1) *(IDX[1]+1)
           if current SHARED < XSHARE then
              current SHARED = YSHARE;
           else if current SHARED = XSHARE then
                current SHARED = XYSHARE;
           fi
           i-th entry of NEW_DR = i-th entry of CURR_DR;
           (i+SCH[0]-1)th entry of NEW_DR= i-th entry of CURR_DR;
           j = j + 2;
        else
           j-th entry of NEW_DR = i-th entry of CURR_DR;
           j = j + 1;
        fi
        increment i;
      od
   fi
  free CURR_DR;
  if ISRES then
```

```
      RES_DRF = NEW_DR;
   else
      DIRECTORY = NEW_DR;
   fi
end split_dr

split_bucket:proc(new_RBN,shared)
   if SHARED  = XYSHARE then
      if ADDR->LEVEL[0] > ADDR->LEVEL[1] then
         SHARED = YSHARE;
      else
          SHARED = XSHARE;
      fi
   fi
   if SHARED = XSHARE then
     call region() to get LOW and UP region boundary in SCALE[0];
     SPLIT_BOUND = ( LOW + UP) / 2;
     get region LOW_ and UP_INDEX in each dimension;

     if UP <= SPLIT_BOUND then
        i,j = 0;
        do while i < NO_REC in BUCKET
          if BUCKET[i].KEY[0] >= SPLIT_BOUND then
             move BUCKET[i] to  NEW_BUCKET[j];
             j = j + 1; fi
             increment i;
        od
             /* adjust mapping of DR to buckets */
        i = region LOW_INDEX[0];
        k = region LOW_INDEX[1];
        do while i <= region UP_INDEX[0]
          do while k <= region UP_INDEX[1]
             PTR = DIRECTORY + i * (SCHEAD[0] - 1) + k;
             if k-th boundary in SCALE[0] < SPLIT_BOUND then
                PTR->CNT = PTR->CNT - j;
             else
                PTR->CNT = j;
                PTR->RBN = BKHEAD.AVAILHEAD; /* new bucket pos */
             fi
             increment PTR->LEVEL[0];
                   /*  adjust SHARED value  */
             get local level with SCALE[0];
              /* get split level by adding one to region level */
             SPLIT_LEVEL = ADDR->LEVEL[0] + 1;
             if LOCAL_LEVEL > SPLIT_LEVEL then
                 do nothing;
             else if LOCAL_LEVEL = SPLIT_LEVEL then
                  if LOW_INDEX[1] = UP_INDEX[1]
                      PTR->SHARED = NOSHARED;
                  else
                      PTR->SHARED = YSHARE;
                  fi
             fi
             increment k;
```

```
            od
            increment i;
         od
         else if LOW >= SPLIT_BOUND then
           perform the same algoritm as above with appropriate
             SCALE and SHARED dimension;
         fi
      else if SHARED = YSHARE then
           perform the same algorithm as above with appropriate
             SCALE and SHARED dimension;
           if UP <= SPLIT_BOUND then
               perform same algorithm as XSHARE case;
           else if LOW > SPLIT_BOUND then
               perform same algorithm;
           fi
      fi

      DR_CHANGED = BK_CHANGED = TRUE;
      if BKHEAD.AVAILCNT > 0 then
           POS = BKHEAD.AVAILHEAD * BKSIZE + sizeof(BKHEAD);
           BKHEAD.AVAILHEAD = the next avail bucket RBN at POS;
           write NEW_BUCKET at POS of BKFD;
           decrement BKHEAD.AVAILCNT;
      else
           if BK_INIT then
               write the very first BUCKET at the next of BKHEAD;
           else
               seek the end of BKFD;
           fi
           write NEW_BUCKET;
           increment BKHEAD.AVAILHEAD;
           increment BKHEAD.BKCNT;
      fi
end split_bucket


split_dr_bucket: proc(shared)
      if SHARED = XYSHARE then
         if RSADDR-> LEVEL[0] > RSADDR->LEVEL[1] then
             SHARED = YSHARE;
         else
             SHARED = XSHARE;
         fi
      fi
      CURR_SIZE = (SCHEAD[0] - 1) * (SCHEAD[1] - 1);

      if SHARED = XSHARE then
        REGION_LEVEL = RSADDR->LEVEL[0];
        call region() to get region LOW_ AND UP_BOUND in RESCALE[0];
        SPLIT_BOUND = (LOW_BOUND + UP_BOUND) / 2;
        get SPLIT_IDX in SCALE[0];
        get region LOW_ and UP_INDEX in each dimension;
        TO_SCALE[1] = T1_SCALE[1] = SCALE[1];
        TO_SCHEAD[1] = T1_SCHEAD[1] = SCHEAD[1];
```

```
   if UP_BOUND <= SPLIT_BOUND then
      allocate T0_SCALE[0] with size of (SPLIT_IDX + 1);
      allocate T1_SCALE[0] with size of (SCHEAD[0]-SPLIT_IDX);
      divide SCALE[0] into two at boundary  SPLIT_IDX;
      copy lower region of SCALE[0] to T0_SCALE[0];
      copy upper region of SCALE[0] to T1_SCALE[0];
      T0_SCHEAD[0] = SPLIT_IDX + 1;
      T1_SCHEAD[0] = SCHEAD[0] - SPLIT_IDX;
      SIZE0 = (T0_SCHEAD[0] - 1) * (T0_SCHEAD[1] - 1);
      SIZE1 = (T1_SCHEAD[0] - 1) * (T1_SCHEAD[1] - 1);
      allocate T0_DR with size of SIZE0;
      allocate T1_DR with size of SIZE1;
      i,j,k = 0;
               /*  divide DIRECTORY along SPLIT_IDX */
      do while i < CURR_SIZE
        if i mod (SCHEAD[0] - 1) < SPLIT_IDX
           copy i-th entry of DIRECTORY to j-th entry of T0_DR;
           j = j + 1;
        else
           copy i-th entry of DIRECTORY to k-th entry of T1_DR;
           k = k + 1;
        fi
        i = i + 1;
      od
            /* adjust mapping of RES_DRF to directory bucket */
      j = region LOW_IDX[1]; i = region LOW_IDX[0];
      do while j <= UP_IDX[1]
         do while i <= UP_IDX[0]
            PTR = RES_DRF + j * (RSCHEAD[0]-1) + i;
            increment PTR->LEVEL[0];
            get LOCAL_LEVEL of i-th interval of RESCALE[0];
            SPLIT_LEVEL = REGION_LEVEL + 1;
            if LOCAL_LEVEL > SPLIT_LEVEL then
               do nothing;
            else if LOCAL_LEVEL = SPLIT_LEVEL then
                   if LOW_IDX[1] = UP_IDX[1] then
                      PTR->SHARED = NOSHARE;
                   else
                        PTR->SHARED = YSHARE;
                   fi
            fi
            if i-th boundary in RESCALE[0] >= SPLIT_BOUND then
                PTR->RBN = DRBKHEAD.AVAILHEAD; fi
            increment i;
         od
         increment j;
      od
   else if LOW_BOUND >= SPLIT_BOUND then
            do same alogrithm but reversing upper and lower
              region of split boundary;
        fi
else if SHARED = YSHARE then
  REGION_LEVEL = RSADDR->LEVEL[1];
  call region() to get region LOW_ and UP_BOUND in RESCALE[1];
```

```
SPLIT_BOUND = (LOW_BOUND + UP_BOUND) / 2;
get SPLIT_IDX in SCALE[1] with SPLIT_BOUND;
get region LOW_ and UP_IDX in each dimension;
T0_SCALE[0] = T1_SCALE[0] = SCALE[0];
T0_SCHEAD[0] = T1_SCHEAD[0] = SCHEAD[0];

if UP_BOUND <= SPLIT_BOUND then
   T0_SCHEAD[1] = SPLIT_IDX + 1;
   T1_SCHEAD[1] = SCHEAD[1] - SPLIT_IDX;
   allocate t0_SCALE[1] with size of T0_SCHEAD[1];
   allocate T1_SCALE[1] with size of T1_SCHEAD[1];
   divide SCALE[1] into two at boundary SPLIT_IDX;
   copy the lower region to T0_SCALE[1];
   copy the upper region to T1_SCALE[1];
   SIZE0 = (T0_SCHEAD[0] - 1) * (T0_SCHEAD[1] - 1);
   SIZE1 = (T1_SCHEAD[0] - 1) * (T1_SCHEAD[1] - 1);
   allocate T0_DR with size of SIZE0;
   allocate T1_DR with size of SIZE1;
   i,j,k = 0;
      /* divide DIRECTORY  */
   do while i < CURR_SIZE
      if i < (SCHEAD[0]-1) * SPLIT_IDX then
        copy i-th entry of DIRECTORY to j-th entry of T0_DR;
        j = j + 1;
      else
        copy i-th entry of DIRECTORY to k-th entry of T1_DR;
        k = k + 1;
      fi
      i = i + 1;
   od
         /* adjust mapping of RES_DRF to directory bucket */
   i = region LOW_IDX[1]; j = region LOW_IDX[0];
   do while i <= UP_IDX[1]
      do while j <= UP_IDX[0]
         PTR = RES_DRF + i * (RESCHEAD[0]-1) + j;
         increment PTR->LEVEL[1];
         get LOCAL_LEVEL of i-th interval in RESCALE[1];
         SPLIT_LEVEL = REGION_LEVEL + 1;
         if LOCAL_LEVEL > SPLIT_LEVEL then
             do nothing;
         else if LOCAL_LEVEL= SPLIT_LEVEL then
                 if LOW_IDX[0] = UP_IDX[0] then
                      PTR->SHARED = NOSHARE;
                 else
                      PTR->SHARED = XSHARE;
                 fi
         fi
         if i-th boundary in RESCALE[1] >= SPLIT_BOUND then
            PTR->RBN = DRBKHEAD.AVAILHEAD; fi
         increment j;
      od
      increment i;
   od
 else if LOW_BOUND >= SPLIT_BOUND then
```

```
            /*  do same algorithm but reversing lower and upper
                region of split boundary */
      fi
   fi

   RSDR_CHANGED = DR_CHANGED = TRUE;
   if DRBK_INIT is TRUE then
      seek DRFD the very next to DRBKHEAD;
      write T0_SCHEAD,T0_SCALE,T0_DR;
      clear the remainder of the directory bucket;
      DRBK_INIT = DRBK_CHANGED = FALSE; fi
   if DRBKHEAD.AVAILCNT > 0 then
     seek DRFD the position of availhead;
     read the next avail directory bucket;
     update the DRBKHEAD.AVAILHEAD with next availhead;
     decrement DRBKHEAD.AVAILCNT;
   else
       seek DRFD the end of the file;
       increment DRBKHEAD.AVAILHEAD;
       increment DRBKHEAD.BKCNT;
   fi
        /*  write new directory bucket */
   write DRFD with T1_SCHEAD,T1_SCALE,T1_DR;
   clear the remainder of the directory bucket;

   free T1_SCALE and T1_DR;
         /*  substitute current dirctory */
   SCHEAD = T0_SCHEAD;
   DIRECTORY = T0_DR;
   SCALE = T0_SCALE;
end split_drbk

merge:proc(bucket, scale, directory)
   /* call function candidate() to find candidate bucket to be
      merged and the dimensioin for merging. The function returns
      TRUE if there is valid candidate and FALSE, otherwise  */
  if candidate(cand_bucket,cand_dim)  is not TRUE then
        message 'no candidate ';
        return;
  fi

  call merge_bucket(cand_bucket,cand_dim)
  if ADDR->SHARED = XYSHARE then
      get full range level F_LEVELk[] in each SCALE[k];
      if ADDR->LEVEL[0] = F_LEVEL[0] and
         ADDR->LEVEL[1] = F_LEVEL[1] then
         call merge_dr(DIRECTORY,IDX[k],XYSHARE);
      else if ADDR->LEVEL[0] = F_LEVEL[0] then
              compute MERGED_SIZE to be after merge;
              if MERGED_SIZE < LOW_THRESHOLD of DRBKSIZE then
                  call merge_scale(SCALE[1],IDX[1],SCHEAD[1]);
                  call merge_dr(DIRECTORY,IDX[k],YSHARE);
              fi
      else if ADDR->LEVEL[1] = F_LEVEL[1] then
```

```
                compute MERGED_SIZE to be after merge;
                if MERGED_SIZE < LOW_THRESHOLD of DRBKSIZE then
                    call merge_scale(SCALE[0],IDX[0],SCHEAD[0]);
                    call merge_dr(DIRECTORY,IDX[k],XSHARE);
                fi
        fi
    fi
end merge

candidate: function(cand_bucket,cand_dim)
        /*   search the first dimension  */
    get lower and upper bound of interval IDX in SCALE[0];
    call region() and get region LOW and UP of the interval;
    REG_BUD_LOW[0] = get_buddy(SCALE[0],LOW,UP,MAXBOUND[0]);
    i = 0;
    do while i < SCHEAD[0]
        if i-th boundary in SCALE[0] =  REG_BUD_LOW[0] then
            PTR = DIRECTORY + YIDX * (SCHEAD[0]-1) + i;
            if (ADDR->CNT + PTR->CNT) < UPPER_THRESHOLD and
                ADDR->LEVEL[k] = PTR->LEVEL[k] in each k then
                TO_BE_MERGED[0] = PTR->RBN; fi
                break;
        fi
        increment i;
    od

    search the second dimension with the same algorithm as above;

    if both TO_BE_MERGED[K] is available then
        if ADDR->LEVEL[0] > ADDR->LEVEL[1] then
            CAND_BUCKET = TO_BE_MERGED[0];
            CAND_DIM = 0;              /*  first dimension  */
        else if ADDR->LEVEL[0] < ADDR->LEVEL[1] then
                CAND_BUCKET = TO_BE_MERGED[1];
                CAND_DIM = 1;          /* second dimension  */
        else if SCHEAD[0] < SCHEAD[1] then
                CAND_BUCKET = TO_BE_MERGED[1];
                CAND_DIM = 1;
        else
                CAND_BUCKET = TO_BE_MERGED[0];
                CAND_DIM = 0;
        fi
    else if TO_BE_MERGED[0] is available then
            CAND_BUCKET = TO_BE_MERGED[0];
            CAND_DIM = 0;
    else if TO_BE_MERGED[1] is available then
            CAND_BUCKET = TO_BE_MERGED[1];
            CAND_DIM = 1;
    fi

    if there is valid CAND_BUCKET then
          return TRUE;
    else
          return FALSE;
```

```
   fi
end candidate

merge_bucket:proc(cand_bucket,cand_dim)
   seek BKFD for the position of CAND_BUCKET;
   read the bucket into TEMP_BK[NO_REC];
   i, j, k = 0 ;
   do while all i,j,k < NO_REC
      do while BUCKET[j].KEY[0] > 0  /* skip record exist */
          j = j + 1;
      od
      do while TEMP_BK[k].KEY[0] < 0  and k < NO_REC
          k = k + 1;
      od
      if  k >= NO_REC then
         break; fi
      BUCKET[j] = TEMP_BK[k];
      increment all i,j,k;
   od
       /*  collect merged bucket at avail list  */
   get total record count, MERGED_CNT;
   clear TEMP_BK[];
   TEMP_BK[0].KEY[0] = BKHEAD.AVAILHEAD;
   write back TEMP_BK at CAND_BUCKET position;
   BKHEAD.AVAILHEAD = RBN of CAND_BUCKET;
   increment BKHEAD.AVAILCNT;
   DR_CHANGED = BK_CHANGED = TRUE;
       /*  adjust mapping of directory  */
   if CAND_DIM = 1 then
      get low and up boundary of interval IDX[1] in SCALE[1];
      call region() with the values and decremented
           ADDR->LEVEL[1] to get region LOW_ and UP_BOUND in
           SCALE[1];
      get low and up boundary of interval IDX[0] in SCALE[0];
      call region() with the values to get region LOW_ and
           region UP_BOUND in SCALE[0];
      get region index LOW_IDX[k] and UP_IDX[k] with above values
      i = LOW_IDX[1];  j = LOW_IDX[0];
      do while  i <= UP_IDX[1]
         do while  j <= UP_IDX[0]
            PTR = DIRECTORY + i * (SCHEAD[0]-1) + j;
            decrement PTR->LEVEL[1];
            PTR->CNT = MERGEC_CNT;
            PTR->RBN = ADDR->RBN;
            if PTR->SHARED < XSHARE then
                 PTR->SHARED = YSHARE;
            else if PTR->SHARED = XSHARE then
                      PTR->SHARED = XYSHARE;
            fi
            increment j;
         od
         increment i;
      od
   else if CAND_DIM = 0 then
```

```
            perform the same algorithm as above base on SCALE[0];
    fi
end merge_bucket

merge_dr: proc(directory,idx[k],merge_dim)
    if MERGE_DIM = XSHARE then
      PRE_MIDX=index(SCALE[0],IDX[0],ADDR->LEVEL[0],MAXBOUND[0]);
      MIDX = PRE_IDX + 1;
      allocate MERGED_DR with decreased size by one column;
          /* copy old directory to new one */
      i,j,k,l = 0;
      do while i < SCHEAD[1]-1
         do while j < schead[0]-1
            if j  !=  P_MIDX then
               copy l-th entry of DIRECTORY to k-th entry of
               MERGED_DR;
               increment k,l;
            else
                l-th entry of DR.SHARED = YSHARE;
                copy l-th entry to k-th entry of MERGED_DR;
                increment j,k;
                l = l + 2;
            fi
            increment j;
         od
         increment i;
      od
      decrement SCHEAD[0];
      free DIRECTORY;
      DIRCTORY = MERGED_DR;
   else if MERGE_DIM = YSHARE then
    PRE_MIDX = index(SCALE[1],IDX[1],ADDR->LEVEL[1],MAXBOUND[1]);
    MIDX = PRE_MIDX + 1;
    allocate MERGED_DR with decresed size by one row;
    i,j,k,l = 0;
    do while i < SCHEAD[1]-1
       do while j < SCHEAD[0]-1
          if i < PRE_MIDX or i > MIDX then
             copy l-th entry of DR to k-th entry of MERGED_DR;
             increment l,k;
          else if i = PRE_MIDX then
             l = l + SCHEAD[0] - 1;
             break;
          else if i = MIDX then
             l-th entry of DR.SHARED = XSHARE;
             copy l-th entry of DR to k-th entry of MERGED_DR;
             increment l,k;
          fi
          increment j;
       od
       increment i;
    od
    decrement SCHEAD[1];
    free DR;
```

```
   DR = MERGED_DR;
   else if MERGE_DIM = XYSHARE then
      allocate MERGED_DR with size of one;
      entry of MERGED_DR = entry of ADDR;
      MIN = lowest boundary of SCALE[0];
      MAX = highest boundary of SCALE[0];
      free SCALE[0];
      allocate SCALE[0] with size of two;
      assign MIN to thefirst entry of SCALE[0];
      assign MAX to the second entry of SCALE[0];
      MIN = lowest boundary of SCALE[1];
      MAX = highest boundary of SCALE[1];
      free SCALE[1];
      allocate SCALE[1] with size of two;
      assign MIN to the first entry of SCALE[1];
      assign MAX to the second entry of SCALE[1];
      SCHEAD[0] = SCHEAD[1] = 2;
      free DR;
      DR = MERGED_DR;
   fi
end merge_dr


update: proc(gridfile)
     /*  update  records interactively */
   CONTINUE = TRUE;
   do while CONTINUE = TRUE
      get RECORD from terminal;
      if (i = find(BKFD,RECORD)) >= 0 then
         get NEW_INFO from terminal;
         BUCKET[i].INFO = NEW_INFO;
         BK_CHANGED = TRUE;
      else
         message 'requested record not exist';
      fi
      get a value for and assign to CONTINUE from terminal;
   od
end update

range_query: proc(gridfile)
     /* get range in each dimension  /*
   get range bounds BEGIN[k] and END[k] from terminal;
   BEG_RS_IDX[k] = index(RESCALE[k],BEGIN[k],RESCHEAD[k]);
   END_RS_IDX[k] = index(RESCHEAD[k],END[k],RESCHEAD[k]);
   allocate RS_QUEUE; /* for keeping serched directory bucket */

   i = BEG_RS_IDX[0];   j = BEG_RS_IDX[1];
   do while i <= END_RS_IDX[0]
      do while j <= END_RS_IDX[1]
         RSADDR = RES_DRF + j * (RESCHEAD[0] - 1) + i;
         if RSADDR->RBN is in RS_QUEUE
             break;
         else
             insert RSADDR->RBN into RS_QUEUE;
```

```
          fi
          if RSADDR->RBN  != currDR_RBN then
               POS = RSADDR->RBN * DRBKSIZE + sizeof(DRBKHEAD);
               call loaddrf(DRFD,POS); /* replace directory */
               currDR_RBN = RSADDR->RBN;
          fi
                  /* do for both of k   */
          if first boundary in SCALE[k] < BEGIN[k] then
               BEG_IDX[k] = index(SCALE[k],BEGIN[k],SCHEAD[k]);
          else
               BEG_IDX[k] = 0;
          fi
          if last boundary in SCALE[k] > END[k] then
               END_IDX[k] = index(SCALE[k],END[k],SCHEAD[k]);
          else
               END_IDX[k] = SCHEAD[k] - 1;
          fi

          allocate QUEUE;  /* for keeping serched data bucket */
          k = BEG_IDX[0];  l = BEG_IDX[1];
          do while k <= END_IDX[0]
             do while l <= END_IDX[1]
                ADDR = DIRECTORY + l * (SCHEAD[0] - 1) + k;
                if ADDR->RBN is in QUEUE then
                     break;
                else
                     insert ADDR->RBN into QUEUE;
                fi
                if ADDR->RBN  != curr_RBN then
                     POS = ADDR->RBN * BKSIZE + sizeof(BKHEAD);
                     call loadbkf(BKFD.POS);
                     curr_RBN = ADDR->RBN;
                fi
                sort records in current BUCKET;
                report records in the RANGE;
                l = l + 1;
             od
             k = k + 1;
          od
          free QUEUE;
          j = j + 1;
       od
       i = i + 1;
    od
    free RS_QUEUE;
end range_query
```

VITA 2

Chang Chun Han

Candidate for the Degree of

Master of Science

Thesis: A GRID FILE APPROACH TO LARGE MULTIDIMENSIONAL DYNAMIC DATA STRUCTURES

Major field: Computing and Information Sciences

Biographical:

Personal Data: Born in Seoul, Korea, February, 1949, the son of Mr. Young Poong Han and Young Shin Song.

Education: Graduated from Pai Chai High School, Seoul, Korea, in February, 1967; received Bachelor of Science degree in Textile Engineering from Seoul National University in February, 1975; completed requirements for the Master of Science degree at Oklahoma State University in May, 1988.

Professional Experience: Sales Engineer in Textile Export Dept., Hyosung Corp., Seoul, Korea, 1975-1978; Sales Engineer in Oversees Plants Project Dept., Hyundai Int'l Corp., Seoul, Korea, 1978-1980; Technical Project Appraiser in Technical Appraisal Office, Korea Long Term Credit Bank, Seoul, Korea, 1980 to present.