DYNAMIC ADDITION AND REMOVAL OF ATTRIBUTES IN BANG FILES

By

NALINI T. HOSUR
//

Master of Arts
Karnatak University
Karnatak, India
1981

Doctor of Philosophy
Karnatak University
Karnatak, India
1981

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, 1991

DYNAMIC ADDITION AND REMOVAL OF

ATTRIBUTES IN BANG FILES

Thesis Approved:

Huizhu Lu

Thesis Adviser

R. E. Haskell

J P Chandler

Norman N Durham

Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my sincere appreciation to Dr. Huizhu Lu, for her encouragement, advice, and patient guidance. I also wish to thank my committee members, Dr. George E. Hedrick and Dr. J P. Chandler for their assistance.

I owe my special thanks to my family, and my friends for their encouragement, and support which made this thesis possible.

TABLE OF CONTENTS

Chapter		Page
I.	INTRODUCTION.	1
II.	BANG FILE DATA STRUCTURES	5
	Overview of Multidimensional Data Structures	5
	Tree Structures.	6
	Multiple Extendible Hashing.	7
	Grid and BANG Files	8
	Structures and Characteristics of BANG Files	10
	Directory.	11
	Update	13
	Splitting	14
	Merging	14
	Queries	15
	Creation of Empty Buckets During Subdirectory Splitting	17
	Creation of Empty Regions During Deletion of Records	22
	Dependence of Directory Entries on the Order of the Data	25
III.	TECHNIQUES FOR DYNAMIC CHANGE OF ATTRIBUTES	30
	Naive Approach	30
	Directory Modification Technique	31
	Addition of Attributes	32
	Attribute Addition in The Presence of Buddies.	33
	Modification of Directory Entries	36
	Transformation of Directory Entries	37
	Transformation of Single Entries	38
	An Example of Transformation of Directory Entries	39
	Removal of The Attributes	40
	Modification of Directory Entries	41
	Transformation of directory Entries	41

Chapter Page

Transformation Single Entries. 42
An Example of Transformation of Directory Entries. 43

Multiple BANG Files. 44
Addition of Attributes 45
Steps to Add Attributes 46
Deletion of Attributes 46
Steps to Remove Attributes. . . 47
Organization of Multiple BANG Files. 47

IV. RESULTS AND DISCUSSIONS 52

Results and Discussions for 'Directory Modification' 52
Results and Discussions for 'Multiple BANG' Files 69
Results and discussions for Range Queries 70
Results and Discussions for Partial Queries 79

V. CONCLUSIONS AND RECOMMENDATIONS 91

BIBLIOGRAPHY. 96

APPENDIX - PROCEDURES TO ADD AND REMOVE ATTRIBUTES . . 99

LIST OF TABLES

Table		Page
I.	Data and Directory Occupancy	28
II.	Range Query For 'abcd'	28
III.	Partial Query For 'a'.	29
IV.	Partial Query For 'abc'.	29
V.	Partial Query for 'cd'	29
VI.	Addition of attribute Test File I (N = 1,000).	55
VII.	Addition of Attribute Test File I (N = 2,000).	56
VIII.	Addition of Attribute Test File I (N = 5,000).	57
IX.	Addition of Attribute Test File II (N = 1,000).	58
X.	Addition of Attribute Test File II (N = 2,000).	59
XI.	Addition of Attribute Test File II (N = 5,000).	60
XII.	Removal of Attribute Test File I (N = 1,000).	61
XIII.	Removal of Attribute Test File I (N = 2,000).	61
XIV.	Removal of Attribute Test File I (N = 5,000).	62
XV.	Removal of Attribute Test File II (N = 1,000).	62
XVI.	Removal of Attribute Test File II (N = 2,000).	63
XVII.	Removal of Attribute Test File II (N = 5,000).	63
XVIII.	Range Retrieval (N = 1,000, 25% Data with 3rd Attribute).	72
XIX.	Range Retrieval (N = 1,000, 50% Data with 3rd Attribute).	72
XX.	Range Retrieval (N = 2,000, 25% Data with 3rd Attribute).	73

Table		Page
XXI.	Range Retrieval (N = 2,000, 50% Data with 3rd Attribute).	73
XXII.	Range Retrieval (N = 5,000, 25% Data with 3rd Attribute).	74
XXIII.	Range Retrieval (N = 5,000, 50% Data with 3rd Attribute).	74
XXIV.	Partial Query for 'ab' (100% Search)	81
XXV.	Partial Query for 'a' (100% Search)	81
XXVI.	Partial Query for 'ac' (50% Search)	82
XXVII.	Partial Query for 'bd' (50% Search)	82
XXVIII.	Partial Query for 'abc' (50% Search)	83
XXIX.	Partial Query for 'cd' (25% Search)	83
XXX.	Partial Query for 'bcd' (25% Search)	84
XXXI.	Range Query for 'abcd' (25% Searrch)	84

LIST OF FIGURES

Figure		Page
2.1.	Nested Block Regions	10
2.2.	Two Level Directory.	12
2.3.	(a) Regions, (b) Directory Entries Before and After Splitting.	13
2.4.	(a) Directory Entries, and (b) Regions Before Splitting	18
2.5.	(a) Directory Entries, (b) Regions, After Splitting (Empty Regions Created)	19
2.6.	After Merging Empty Region	21
2.7.	Merging of Directory Entries During Deletion of Records	23
2.8.	Directory Entries at the End of Deletion of Records	24
2.9.	Partitioning for the Data Arriving in Order 'A'	25
2.10.	Partitioning for the Data Arriving in Order 'B'	27
3.1.	Distribution of Set of Points in 2-D and 3-D	33
3.2.	Splitting of Region <3,2> in 2-D	33
3.3.	Splitting of Region <3,2> in 3-D	34
3.4.	Buddies in 2-D, and 3-D Due to Partition on Y-axis	35
3.5.	Transformation of Directory Entries During Addition of Attribute	39
3.6.	Transformation of Directory Entries During Removal of Attribute	43

Figure Page

3.7. Addition of Attribute 'c' 45

3.8. Before and After Deletion of Attribute 'b' from record #3 47

3.9. Record Distribution in 'Traditional', and 'Multiple BANG' Files 48

3.10. Organization of Multiple BANG files 50

4.1. Dependence of Normalized Disk Accesses on Records Modified - Attribute Addition 64

4.2. Dependence of Normalized Disk Accesses on Records Modified - Attribute Addition 65

4.3. Dependence of Normalized Disk Accesses on Records Modified - Attribute Deletion 66

4.4. Dependence of Normalized Disk Accesses on Records Modified - Attribute Deletion 67

4.5. Average Disk Accesses for Range Query (3-D) Traditional Vs Multiple BANG Files 76

4.6. Average Disk Accesses for Range Query (3-D) Traditional Vs Multiple BANG Files 77

4.7. Average Disk Accesses for Range Query (3-D) Traditional Vs Multiple BANG Files 78

4.8. 'Traditional' and 'Multiple BANG' Files Used for Partial Query 80

4.9. Dependence of Disk Accesses for 3 Search Sizes - 2 Attribute Partial Query 86

4.10. Dependence of Disk Accesses for 2 Search Sizes - 3 Attribute Partial Query 88

4.11. Dependence of Disk Accesses on the Number of Attributes - 25% Search Size 89

CHAPTER I

INTRODUCTION

Database applications with large numbers of records often are stored so that major portions of the data reside on secondary storage. In many existing systems, access to secondary storage is relatively slow compared to operations on data in main memory. As a result the time used for retrieval of data from secondary storage can become the dominant factor in determining the performance of the database system. The BANG (Balanced And Nested Grid) file structure has proved to be an efficient approach to minimizing the number of accesses to secondary storage in response to queries, or for addition and deletion of records with multiple attributes.

Many practical applications however require an added capability: the addition and removal of attributes from an existing database either for some of the records, or all the records. Two simple (hypothetical) examples will illustrate the necessity of addition and removal of attributes.

A physician has information about his patients, which is built into a database. Within the records are the name and address of his patients, some of whom need to be

watched: they are prone to heart disease or similar threat. The attributes (keys) consist of a number of factors such as age, height, weight, cholesterol level etc. When the database was initially created, there was no key corresponding to the ratio of low density lipoproteins to high density lipoproteins, because it was not deemed as an important factor. Recent research has disclosed that this is a critical factor in determining propensity to heart attack. In addition there are other attributes which are now known to be important in determining the level of risk of a heart attack. It is therefore necessary to add these attributes to the database so that a better evaluation of risk can be made.

A second example concerns a database set up by a dealer who sells personal computing systems and work stations. The record consists of the make of hardware; the attributes are the various features available on each system. Since the manufacturers constantly change the features available on each system, it is necessary to add or remove attributes corresponding to features that are added or not provided with the system.

These simple examples clearly illustrate that the ability to add and remove attributes is an important step in extending the utility of a database. If we look upon the ability to add and remove records from a database as a first step in generalizing a static data structure, addition and removal of attributes represents an equally

important second step.

For this reason, all recent relational databases allow the addition and removal of attributes. However implementing this capability in main memory may not be optimal for databases resident on secondary storage devices.

In view of the established superiority of the BANG file structure [FREE 87, and LIAN 88] in minimizing accesses to secondary storage, it is highly desirable to investigate the efficiency of this structure with respect to the addition and removal of attributes. A naive and obvious approach is to change the contents of the entire database whenever we add or remove attributes from any record. However this can be highly inefficient and can seriously limit the applicability of BANG files in many practical applications. To date there is no published work dealing with efficient approaches for the addition and removal of attributes from BANG files. The goal of this thesis is to explore two new techniques for the removal and addition of attributes within the structural framework of BANG files.

In Chapter 2, we give a brief outline of the various approaches towards multiple attribute data structures that resulted in the development of BANG files, and describe in brief the basic workings of the BANG file data structure. In chapter 3, we describe two new approaches for adding and removing attributes from a BANG file; the first of these

approaches is termed the "Directory modification technique", and the second is termed the "Multiple BANG file technique". In Chapter 4, we present results for each of these methods when we add and remove attributes; comparisons are made with the naive approach. In Chapter 5, we analyze the advantages of each of the methods, and conclude with recommendations regarding the use of these methods.

CHAPTER II

BANG FILE DATA STRUCTURES

The BANG file structure represents an important step in the development of efficient data structures for multidimensional data that are accessed with multiple keys. In this chapter, we place the BANG file in context by reviewing other multidimensional data structures. We also provide here details regarding the structure and organization of BANG files so as to be able to refer to them in later sections.

Overview of Multidimensional Data Structures

A number of different data structures have been developed for the storage, retrieval and query of records that have only one attribute per record. A simple extension of these data structures to retrieve records with multiple attributes results in structures that are not efficient in terms of accesses to secondary storage, particularly for interactive queries. There have been many attempts to overcome these deficiencies by designing new multidimensional file structures and access mechanisms. Research has followed three distinct lines [FREE 87]:

1. Tree structures: a generalization to n dimensions.

Some examples are quad trees, quintary trees, polygon trees, range trees, k-d trees, k-d B trees, multiple attribute trees (MAT), multidimensional B trees (MDBT).

2. Multidimensional Extendible Hashing.
3. Grid files and BANG files: a geometrical approach.

Tree Structures

Quad trees [FINK 74] are generalizations of binary trees, and deal with two dimensional data. Each node stores one record and has up to four offspring, each a node. Both insertion and region searching are quite efficient, but deletion and merging can be complicated. Though the basic concept involved in quad trees can be generalized to an arbitrary number of dimensions, for a m dimensional datum, each node will have 2^m offspring. This results in a large number of pointers, and in the case of the leaf node, many null pointers.

Quintary tree [LEE 80] implementation is limited to data with fewer than 5 attributes. Quintary trees also require that a number of assumptions be made regarding the ordering of the keys in a record, and key specifications in a query.

A k-dimensional binary tree(k-d tree)[BENT 75, BENT 79] is a natural generalization of the standard one - dimensional binary search tree. For the one dimensional tree only one key is used as the discriminator, while for the k-dimensional tree, at each internal node exactly one

of the k keys is used as a discriminator. This structure has certain shortcomings in dealing with large and dynamic databases.

The multiple-attribute-tree (MAT) structure, k-d trees and their variants, and quintary trees fare best when it is necessary to deal with relatively complex queries. However the basic shortcoming of these methods is that they are limited to fairly static databases. An expensive reconstruction is essential for MAT organization when changes occur dynamically.

The Multidimensional B-tree (MDBT) [OUSK 81,SCHE 82] is a method for multiple attribute indexing which uses a B -tree to maintain the filial sets at each level, and imposes an ordering on the filial sets in order to ensure efficient searching. Updates can be done in logarithmic time in the worst case. However this structure is not efficient when there are very small filial set sizes, especially in the last level.

Multiple Extendible Hashing

Hash transformations have access time of O(1). Hashing schemes have been adapted to dynamic files on secondary storage devices by a technique of attaching overflow buckets whenever needed, thus slowly changing the O(1) access time characteristic of hashing towards O(n) time. Radix search trees, known to have faster access than other types of search trees [FRED 60], can be used only for

small files, since they often waste memory. These two addressing schemes are unrelated, but the two goals, namely

1) to make hash tables extendible so that they can adapt to dynamic files;
2) to fill radix search trees uniformly, so that they remain balanced;

gave rise to extendible hashing. In extendible hashing the user is guaranteed no more than two page faults to locate the data associated with a given unique key [RONA 79].

Interpolation-based index maintenance [BURK 83] is a multidimensional extension based on linear hashing [LITW 80]. It extends the hash file organization using chaining for overflow. In case of non-uniform distribution, overflow chains become long and reduce the performance [TAMM 83].

Grid And BANG Files

The grid file [BURK 83, NIEV 84, FREE 87] is a multidimensional data structure that supports multikey access. All keys are treated symmetrically, therefore, queries that involve different keys are processed with equal efficiency. The grid file handles proximity queries such as range and neighbor queries in multidimensional data efficiently [HINR 85]. Query efficiency of this structure tends to increase with large databases compared to k-d-B trees [SARI 87].

In a grid file structure the directory is stored on

secondary memory. This structure has proved to be superior to conventional structures such as inverted files and multilists in terms of adaptation to dynamic environments. The fundamental weakness of grid files is that as the data distribution becomes less uniform, the ratio of directory entries to data buckets increases and the directory expansion approaches an exponential rate. Most of the directory entries point to empty block regions and the problem is magnified by the number of dimensions of the file [FREE 87].

Balanced and Nested Grid file (BANG) [FREE 87, FREE 89a] is of the grid file type. Its most important feature is that, in contrast to the grid file design, its directory always expands at the same rate as the data, whatever the data distribution may be. BANG files become increasingly superior in terms of the rate of directory expansion and the efficiency of access operations, as the data distribution becomes less uniform and/or the number of dimensions increase [HINR 85]. Every direct representation of a sub-space is guaranteed to be a minimal representation, and this substantially improves the efficiency of range, partial, and joins [FREE 89a]. Simulation results which support the above statements can be found in [LIAN 88]. Previous studies have established that the BANG file structure is well suited for large databases with multiple attributes per record.

Structure And Characteristics Of BANG Files

Although the BANG file is of the grid file type, it is sufficiently different in its characteristics and performance to distinguish itself from earlier grid file structures. In this section, we describe in brief the major features of BANG file structure; the reason is that we will need to refer to these details in subsequent chapters and by describing them in this chapter, we will avoid redescribing them in later chapters.

Each directory entry of the BANG file is a unique number pair <r,l> where r is region number and l is level number. There is a one to one correspondence between a directory entry and a data bucket. This is made possible by allowing nested block regions ie., if two subspaces into which data space has been partitioned intersect, then one of the subspaces completely encloses the other. In Figure 2.1 region R1 encloses R2.

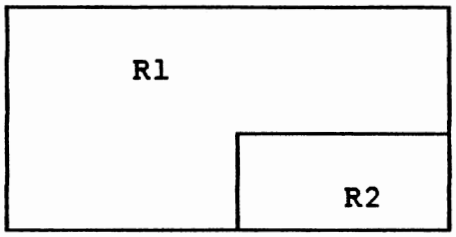


Figure 2.1. Nested Block Regions

The subspaces do not have to be hyperrectangles. In addition to this, there is a particular numbering scheme used for BANG files that allows us to locate the entries unambiguously. The binary numbering scheme is simpler and provides the following properties:

1. For a given set of keys (k1,k2...kn) allows us to calculate the smallest region number in which they lie.
2. Each region number is identified by a unique number pair <r,l> where r is the region number and l is the level
3. It is possible to find all enclosing regions at possible levels for a given region number at a certain level.

Directory

As described before each directory entry of the BANG file structure is number pair <r,l>, where r is region number and l is level number. Only a one dimensional array is necessary to constitute the directory. But for very large databases the directory itself may become too large to fit into primary memory, and hence must be stored on the disk. It is essential that directory be organized efficiently.

A two level directory is maintained. The root directory resides in the main memory and points to a subdirectory bucket. The subdirectory is the second level directory, which resides on the secondary memory and manages the data buckets, see Figure 2.2. In both the root as well as the secondary directory the entries are arranged

in order of increasing partition level. This avoids the ambiguity in locating an entry.

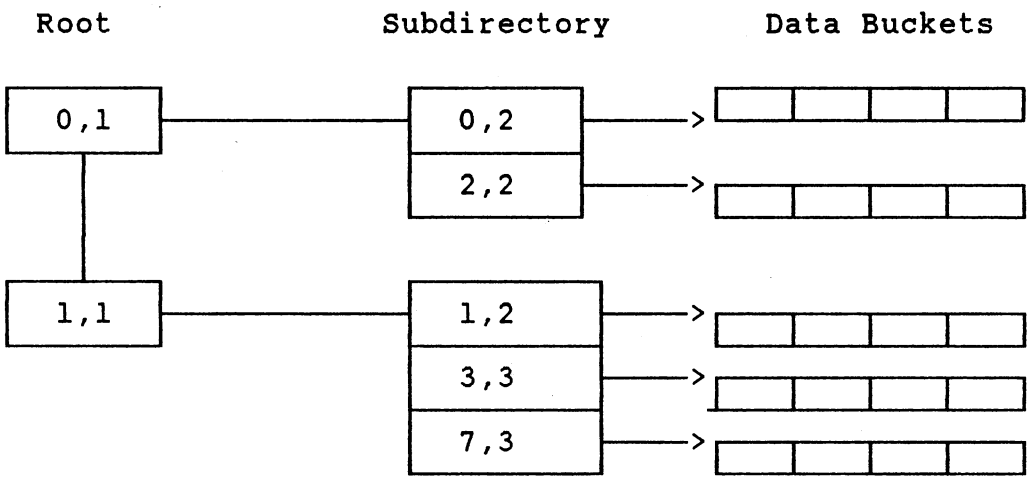
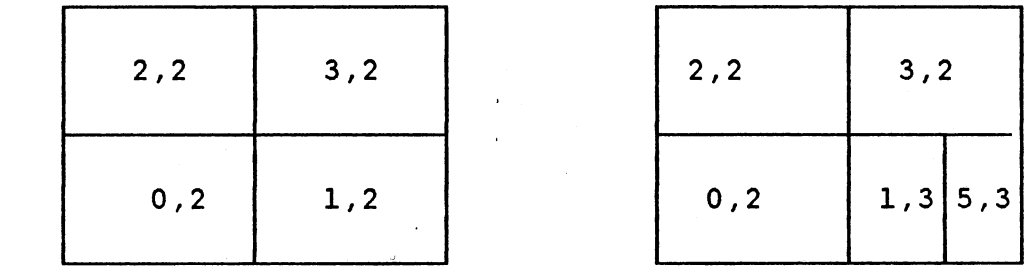
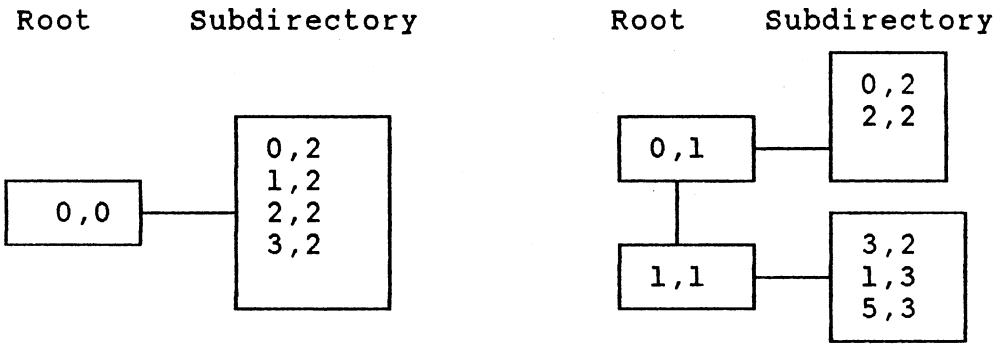


Figure 2.2. Two Level Directory

The complete BANG file directory is organized as a tree structure. When the first leaf node overflows it is split into by using the same method applied to the data bucket. Block regions are treated as data points themselves. Figure 2.3 shows the splitting of subdirectory and its effect on the root. Let the subdirectory capacity be 4. When one of the entries <1,2> is split into (1,3>, and <5,3>, the original root entry <0,0> needs to be partitioned, thus gets split into <0,1>, and <1,1>. Figure 2.3b illustrates entries corresponding to Figure 2.3a.



(a) Regions



(b) Directory Entries

Figure 2.3. (a) Regions (b) Directory Entries Before and After Splitting

Update

The addition of a record may cause the capacity of a bucket to exceed. In which case a split operation is performed. The deletion of a record may cause the number of records in a bucket to fall below a threshold level. In that case it may be possible to merge with another bucket to economize disk space. These two operations are described below.

Splitting

When a data bucket overflows, the corresponding logical region is partitioned into two, with on enclosing the other. The region containing more data points is halved until the 'best balance' [FREE 87, FREE 89a] is achieved, then the best one is chosen. Assuming that there is no preferred attribute, a pre-defined cyclic partitioning through all the dimensions is continued.

1. If the balance is achieved at the first division, then buddy regions are created. The directory entry <r,l> is replaced by two entries <r,l+1>, and <r+2^{l},l+1>.
2. If number partitions is greater than one, the external boundary of the partitioned logical remains the same, but a new logical region is created within it. The two entries are <r,l>, and <r_1,l+n>, where r_1 is the newly formed region, and n is number of partitions.
3. Splitting of a region is always treated as a continuation of an upper level split which sometimes allows distribution without creating new data buckets [FREE 87, FREE 89a].

Merging

A very simple and logical strategy is chosen for merging. The three possibilities of merging are:

1. Merge the logical region with a region that it immediately encloses (starting with the smallest).

2. Merge with the immediately enclosing region.
3. Merge with a buddy.

Merging with the immediate enclosed or enclosing region prevents the ambiguity in the directory entries, which are adjusted after each merging operation.

Queries

1. Exact query: in which values of indexed attribute is specified.
2. Partial match query: values of d < n of the indexed attributes are specified.
3. Range query: range is specified for each attribute.

1. Exact match query

Let k1,k2....kn be the values of n attributes. In order to locate a record with these values, using the mapping function calculate the region r, that encloses the record at the current level l. Using the identifier <r,l> the record P is located/searched in the following steps.

i. Search the root directory for an entry $<r_1,l_1>$, where is the smallest region that encloses the region <r,l>.

ii. Retrieve the subdirectory bucket pointed by the root entry $<r_1,l_1>$.

iii. Search the subdirectory bucket for an entry $<r_2,l_2>$, where r_2 is the smallest region that encloses <r,l>.

iv. Retrieve the data bucket pointed by <r_2,l_2>, and search for the record P.

In the BANG file structure exact queries, whether successful or unsuccessful should require no more than two disk accesses.

2. Partial Match query:

For a partial match query, the key values k_{i1},k_{i2},..k_{im} are such that i1 < i2 < ....im and im< n , where n is the number of attributes. The query is treated in exactly the same way as the exact match query, except the given key values are transformed into a set of region identifiers.

i. Pick one region identifier from the set and use the same method as in exact match query to locate the data bucket corresponding to it.

ii. Find all the records in the data bucket that fulfil the query.

iii. Carry on step i, and ii until the set is empty.

In order to minimize the disk accesses the same directory and data bucket are loaded into memory exactly once.

3. Range query:

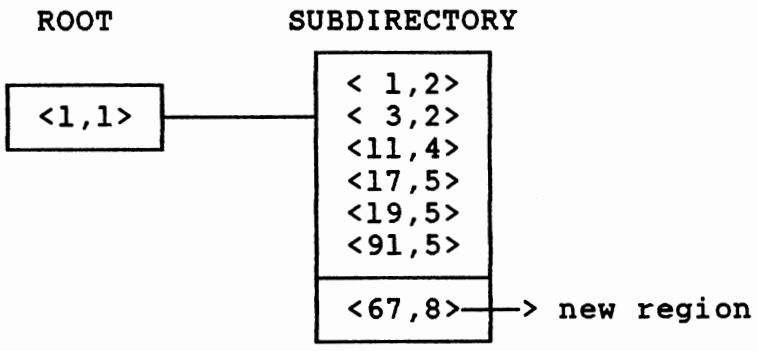
The range query is of the form of Li <= Ai <= Ui, where Li and Ui are the lower and upper bounds for an attribute Ai. The method is exactly same as the partial match query.

This completes a description of the details of the

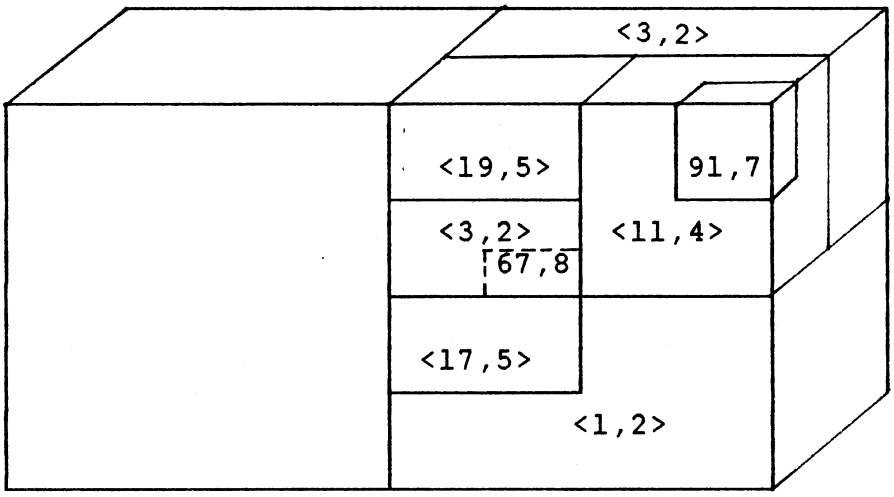
BANG file structure which are well covered in the article by [FREE 87]. However, in the process of devising new approaches for the addition and removal of attributes, several interesting features of the behavior of the BANG file structure have been observed. These features have not yet been described in the existing literature, but can greatly increase the difficulty in developing general methods for the addition and removal of attributes.

Creation of Empty Buckets During Subdirectory Splitting

Let the subdirectory capacity be 6. A seventh entry/region causes the subdirectory to split and in turn changes the root entires. Consider the subdirectory entries and regions pointed by a root entry <1,1> Figure 2.4. Due to the new entry <67,8>, root <1,1> is split into entries <1,1> and <3,4> Figure 2.5.

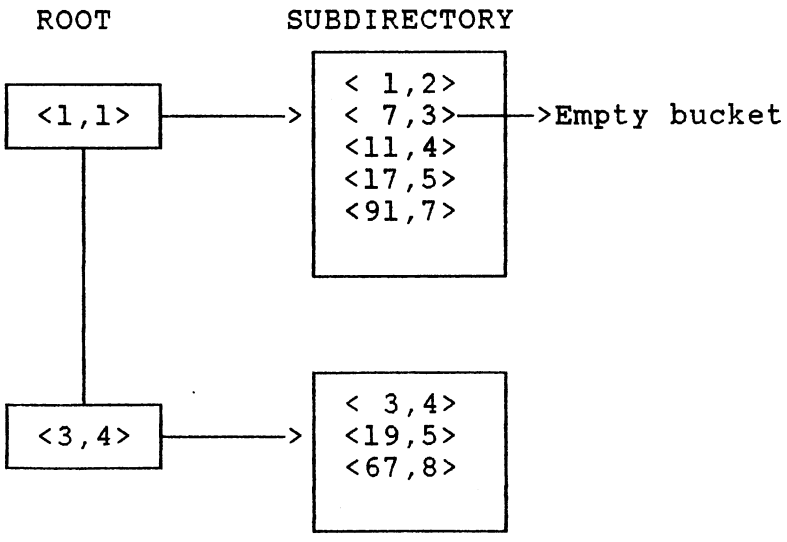


(a). Directory Entries

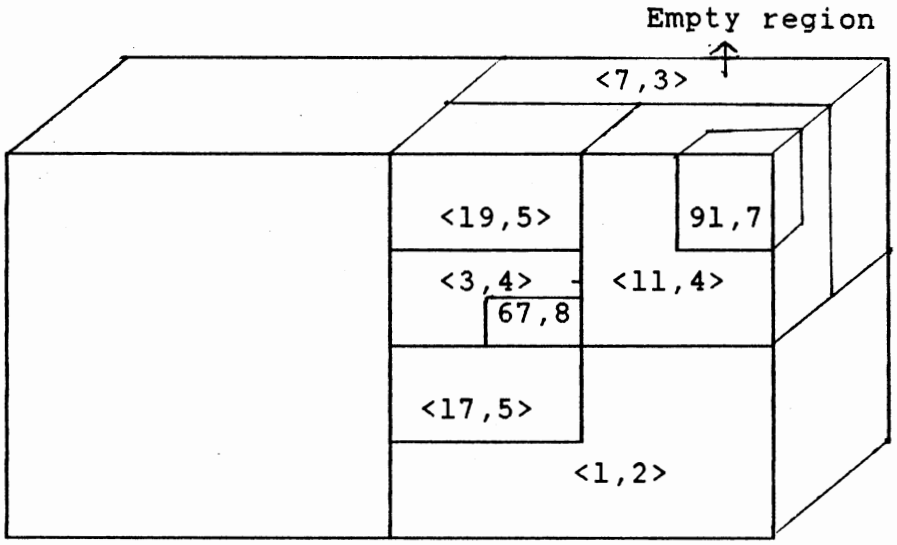


(b). Regions Before Splitting

Figure 2.4. (a) Directory Entries, and (b) Regions before splitting



(a). Directory Entries



(b). Regions

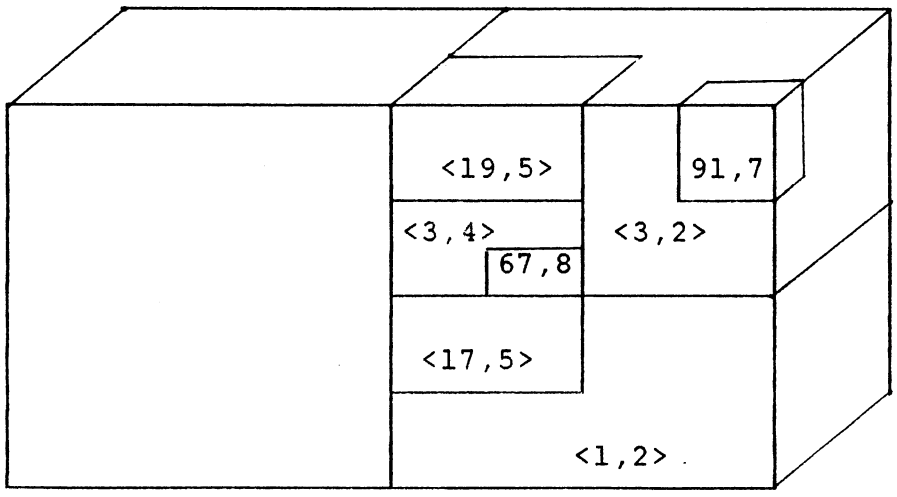
Figure 2.5. (a) Directory Entries, (b) Regions After Splitting (Empty Region Created)

During the splitting, portions of the region <3,2> get divided into newly created roots: <3,4> into root <3,4>, and <7,3> into root <1,1> respectively. When the data is extremely non-uniform it may happen that one of these regions may be empty. In the case, where <7,3> is empty, this needs to be merged with some region, using the merging strategy. The target region with which the empty region can be merged either is enclosed immediately, a buddy, or the enclosing region. Unfortunately, none of these regions are present in the same directory.

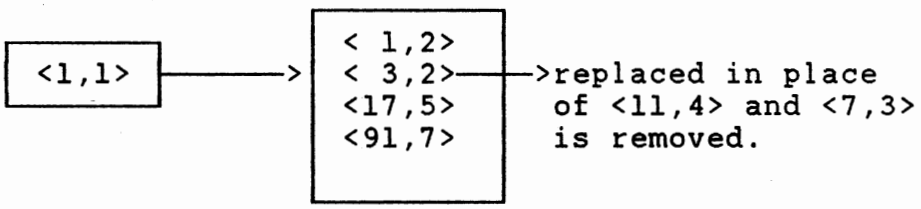
It is contradictory to have empty regions in the BANG file structure. The author proposes a new strategy to merge such empty regions.

Look for the **siblings** of the empty region, i.e children of its parent. In the current example, <7,3> Figure 3.5 is the empty region, and <3,2> is the parent. Any region that is enclosed by <3,2> is called its child. Any such child that is in the same bucket as that of the empty bucket can be chosen to be merged with.

In the above example, Figure 2.5b we can merge <7,3> with <11,4>. Rename the area covered by these two as <3,2> Figure 2.6a. Delete the entry <7,3> replace entry <11,4> by <3,2> Figure 2.6b. This strategy allows us to have better directory occupancy.



(a). Regions



(a) Directory Entries

Figure 2.6. After Merging Empty Region

Steps involved in merging the empty region:

- Locate the sibling in the same subdirectory.
- Replace the sibling by an entry, which was the common parent to both the empty and the sibling *.
- Delete the entry corresponding to the empty region.

Another point that needs to be addressed during directory splitting is that regions cannot be considered as points (as in the bucket splitting). Unlike points,

---

* Consider the binary representation of the two regions being merged.

<7,3> <11,4>

1 1 1 1 0 1 1

Up to level 2, they have the same parent, which is 3. This is decided by looking at the common bits starting from the least significant position. Thus in the above example, <11,4> is replaced by <3,2>.

regions have boundaries, and may be spread into two directory buckets (as explained in the above example). At every step of balancing it is necessary to see if any region gets divided into two directories, and if it does, ensure that there are no empty regions.

Creation of Empty Regions During The Deletion of Records

It is not unusual to see some empty buckets during deletion. Up to a point the empty buckets can be merged with another region according to the merging strategy. But at some point merging is no longer possible. In the example, Figure 2.7, there are 46 buckets in the subdirectory. At the time of deletion, as the capacity of a bucket falls below threshold it gets merged, Figure 2.8. At the end of deletion, there are 39 buckets. Out of these 7 are empty (about 17.94%); further merging is not possible.

As suggested before, these empty buckets can be merged with their siblings. Another possibility is to convert all the buddies into enclosing regions; then it is possible to find an enclosing region. After merging we convert back into buddies if possible.

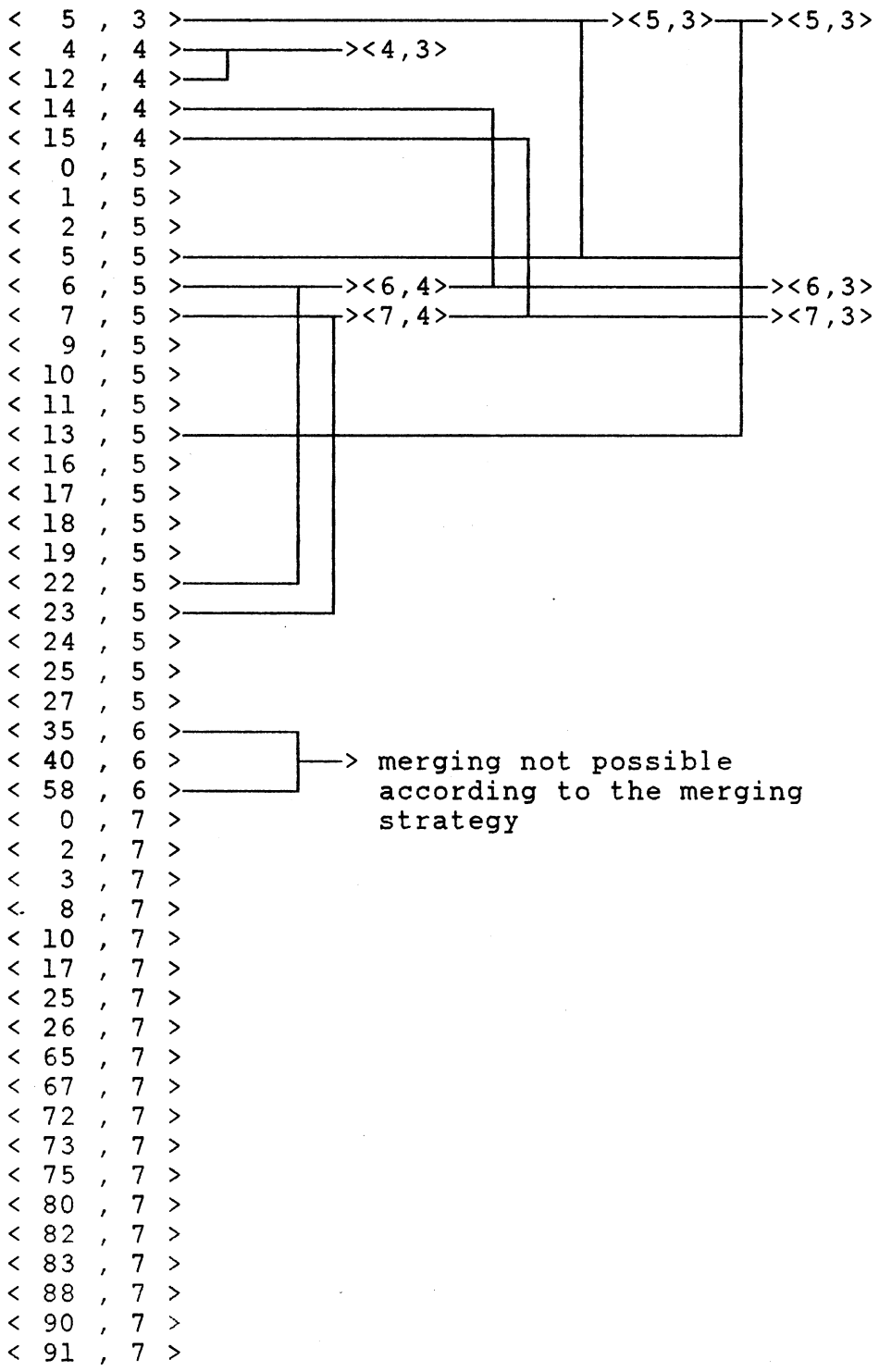


Figure 2.7. Merging of Directory Entries
During the Deletion of Records

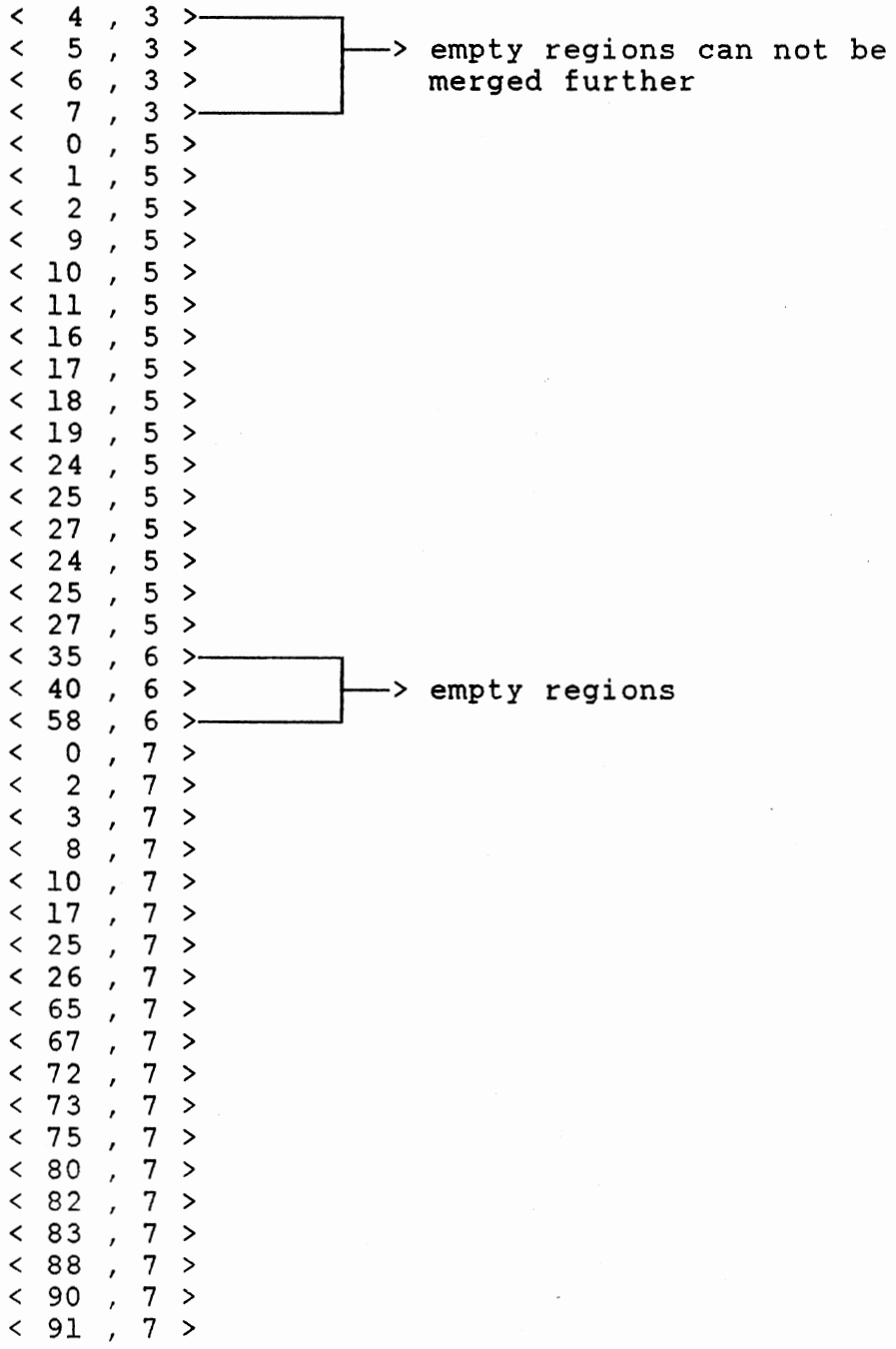
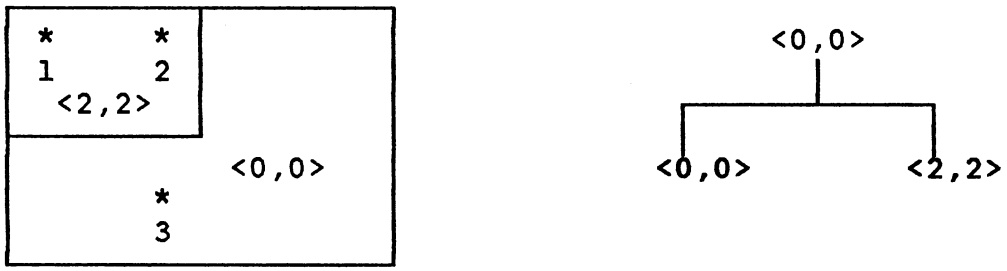


Figure 2.8. Directory Entries at the End of Deletion of Records

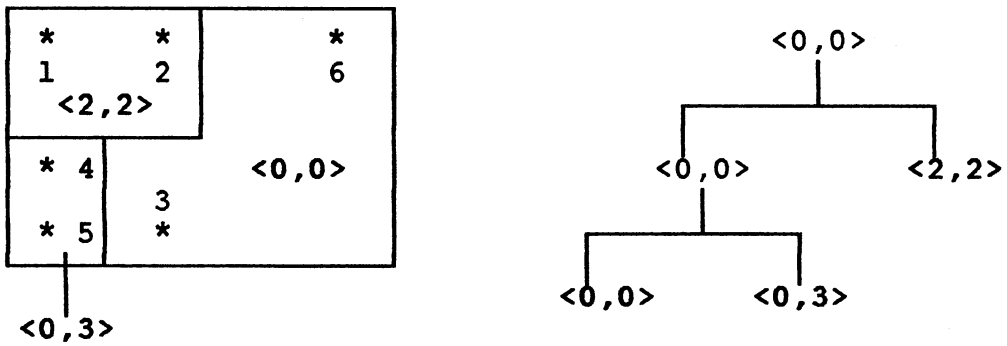
Dependence Of Directory Entries On The Order Of The Data

An interesting feature of the BANG File structure is that a permutation of a data set can result in different entries in the directory; both sets of directory entries correctly represent the same data. Thus the BANG file directory entries are NOT UNIQUE FOR A GIVEN SET OF DATA. This is shown below.

Let the data bucket capacity be 2 and total records be 6. Consider the situation, when data arrives in order 'A': 1, 2, 3, 4, 5, 6 , Figure 2.9.



(a) First Split

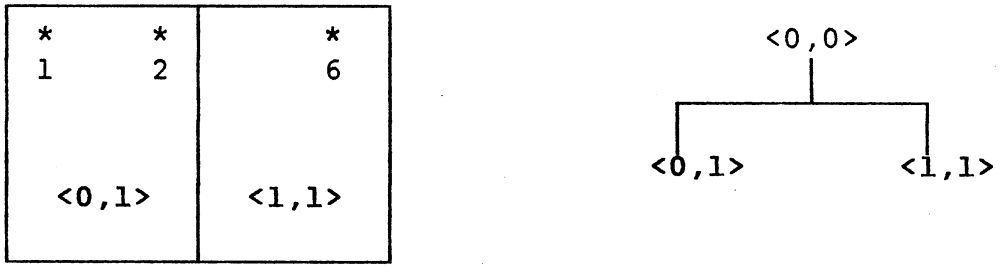


(b) Second Split

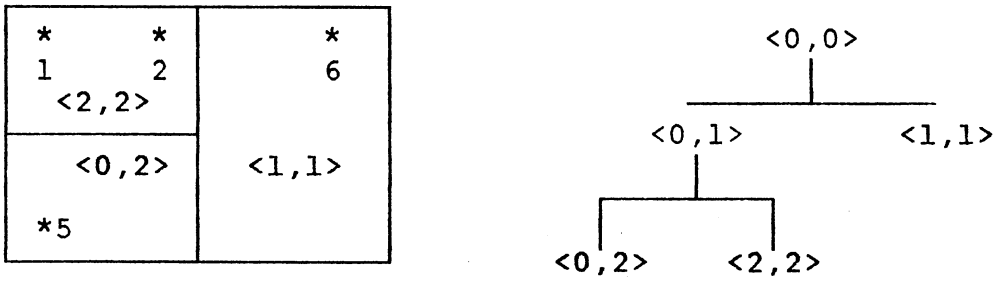
Figure 2.9. Partitioning for a Data Arriving in Order 'A'

When the points are considered in the above order, 3 regions are created, <0,0>, <2,2>, and <0,3> in all.

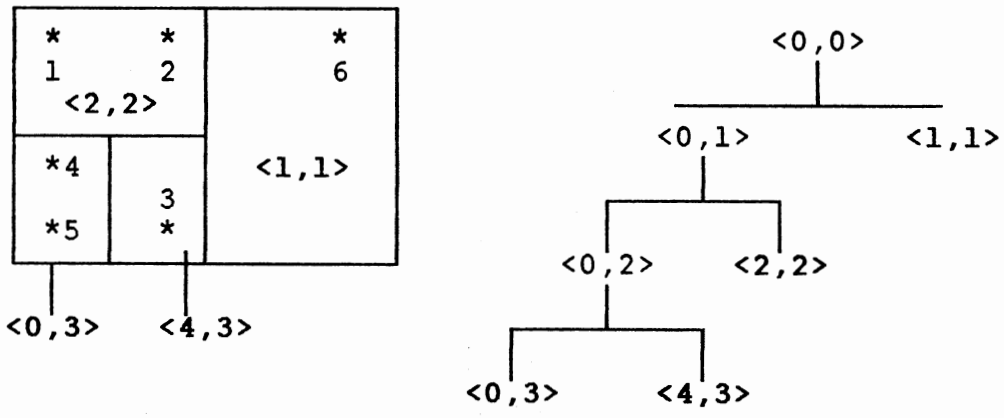
Consider the above records arriving in a different order 'B': 1, 2, 6, 5, 3, 4 Figure 2.10.



(a) First Split



(b) Second Split



(c) Third Split

Figure 2.10. Partitioning for The Data Set Arriving in Order 'B'

When the data points arrive in order 'B' we have 4 regions, as opposed to three with order 'A'.

The dependence of the directory on the order of the data was tested with a simulated data example. Shuffling of the data set was achieved by a random permutation of a single data set, with parameters listed below. The results of the data permutation on data and directory occupancy are listed in Table I, while the number of disk accesses for range and partial queries are compared in Tables II - V. The comparison confirms the earlier results.

Dimension	4	(Attributes 'a','b','c','d')
Total records	5,000	
Data Bucket Capacity	32	
Directory Bucket capacity	8	

TABLE I

DATA AND DIRECTORY OCCUPANCY

	Data Occupancy	Directory Occupancy	Disk Accesses
Before Shuffling	22.12	6.10	20,881
After Shuffling	22.22	5.62	20,801

TABLE II

RANGE QUERY FOR 'abcd'

Range Size	10%	30%	50%
Records Found	0	11	76
Disk Accesses Before shuffling	3	26	124
Disk Accesses After Shuffling	3	24	119

TABLE III

PARTIAL QUERY FOR 'a'

Range Size	10%	30%	50%
Records Found	502	1490	2515
Disk Accesses Before shuffling	50	116	185
Disk Accesses After Shuffling	54	125	191

TABLE IV

PARTIAL QUERY FOR 'abc'

Range Size	10%	30%	50%
Records Found	3	70	302
Disk Accesses Before shuffling	6	33	125
Disk Accesses After Shuffling	7	38	131

TABLE V

PARTIAL QUERY FOR 'cd'

Range Size	10%	30%	50%
Records Found	10	105	309
Disk Accesses Before shuffling	115	143	229
Disk Accesses After Shuffling	108	147	228

CHAPTER III

TECHNIQUES FOR DYNAMIC CHANGE OF ATTRIBUTES

As mentioned in the introduction, the ability to add and delete attributes from an existing data base is an important requirement for most practical problems. The BANG file is known to be an efficient structure for very large databases because it minimizes accesses to secondary storage. However, without an efficient method for addition and removal of attributes, the practical utility of BANG files is severely limited. Before describing new approaches for addition and removal of attributes, we will describe an intuitively obvious approach for performing this task which we term the "Naive approach".

Naive Approach

An attribute (or a key) in a multidimensional database can be associated with a dimension. An N attribute database can be considered as a hyperrectangle in an N - dimensional space. The number of attributes determines the dimensionality of the problem.

The BANG file structure provides a mapping function between the buckets containing the data records, and

regions in N dimensional space that contain the data values. Algorithms for existing BANG files consider the number of attributes fixed. Changing attributes for some or all records requires the creation of a new BANG file, with the new (modified) number of attributes which requires rewriting the data in every data bucket. Since these data buckets are usually resident on secondary storage, this approach is heavily penalized in data access time. The shortcomings of this method are particularly evident when the number of attributes is altered for only a small number of records.

Alternate approaches for the addition and deletion of attributes that are more efficient in terms of accesses to secondary storage. To distinguish the two new methods from the existing approach, the 'naive approach' or the single BANG file approach will be referred to loosely as the 'Traditional' approach.

Directory Modification Technique

When an attribute is added or removed there is a change in number of dimensions; under certain assumptions (which will be explained later in this section), it is possible to modify the directory entries without redistributing the records. This leads to significant savings in disk access time over the 'Traditional' approach. For certain special simple instances, the directory modification approach is not difficult to

implement; however, for more general problems, the structure of the BANG file introduces serious complications. A number of problems occur at almost every stage, and it is necessary to develop individual approaches to handle these problems which make this a highly involved and complex approach. An explanation of the source of these problems with examples, and methods to overcome the difficulties appear below.

It is useful to consider addition and deletion of attributes separately.

Addition Of Attribute/s

In order to illustrate the basic idea, it is helpful to consider a special, simple problem where the addition of an attribute can be done without undue complications. A database with two attributes can be represented as a rectangle in a 2-D space, say the X-Y plane. The addition of a third attribute changes the representation to three dimensions, X,Y, and Z. The 2-D distribution forms the X-Y plane in the 3-D case, which is shown in Figure 3.1. All the points in the X-Y plane in 2-D still lie in X-Y plane in 3-D, except that the values of the Z coordinate are zero.

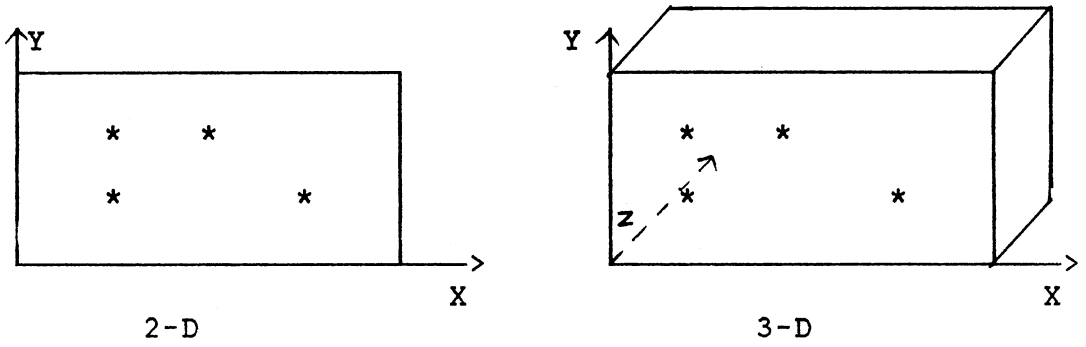
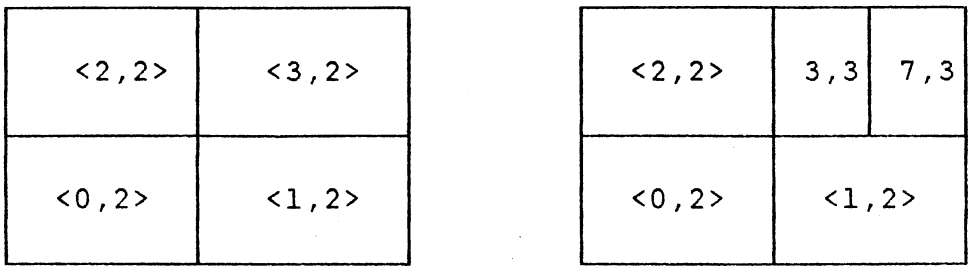


Figure 3.1. Distribution of Set of Points in 2-D and 3-D

Attribute Addition In The Presence Of Buddies

We will consider next a problem where the above simple approach cannot be employed.

Figure 3.2 shows the data distribution in a 2-D database. Let us consider the situation, when the region <3,2> Figure 3.2a, is split into buddies, and the corresponding partitions for 3-D.



(a) Before Splitting (b) After Splitting

Figure 3.2. Splitting of Region <3,2> in 2-D

In 2-D, after partition on the X axis, the entry <3,2>, results in buddy regions <3,3>, and <7,3> Figure 3.2b. But in 3-D, Figure 3.3 due to the extra partition on the Z axis, the entry <3,2> results in enclosing regions <3,2>, and <11,4> Figure 3.3b.

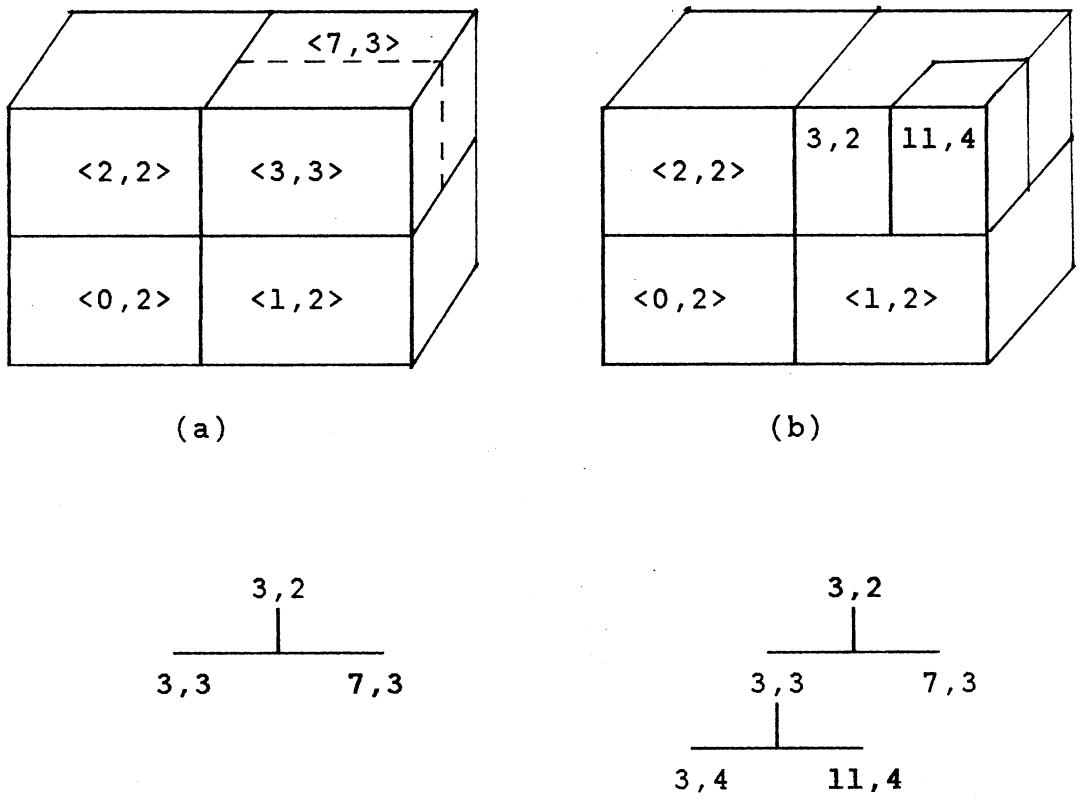


Figure 3.3. Splitting of Region <3,2> in 3-D

On the other hand the corresponding entries of <7,4>, and <15,4> have remained buddies, <11,5>, and <27,5> respectively, as shown in the Figure 3.4.

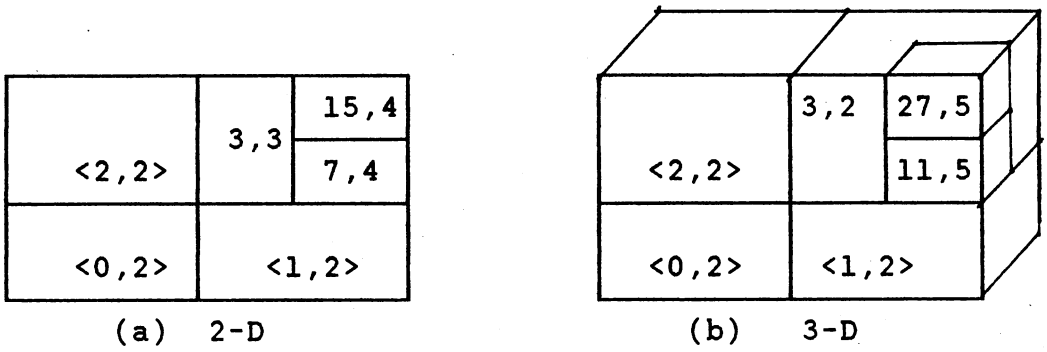


Figure 3.4. Buddies in 2-D and 3-D due to Partition on Y-axis

The reason for this is that balance was obtained in the first partition on the Y-axis in both the cases.

The increase in the level is due to the additional split in the Z direction at the end of every X,Y cycle.

The region number is controlled by the bit settings associated with each partition level. The addition of an attribute changes the partition sequence (zyxzyx instead of yxyx) and consequently alters the bit settings in the region. For simple modifications, that do not involve buddies a transformation function can be derived to account for the changes in the dimensionality of the database. As the levels increase, transformation from one dimension to another requires more thought. For certain levels the buddy regions in 2-D may not remain buddies in 3-D as pointed out in the example before. This is due to the extra partition on the Z axis. In 2-D at the end of every

cycle i.e. after a partition on Y-axis, the next partition is on the X-axis. If there is a balance after a first partition on X-axis, the resulting regions become buddies. If the same data is considered in 3-D, after the Y partition, it has to go through the Z partition and then the X partition, to achieve the balance. Since this requires more than one partitioning, it results in enclosing regions.

It is clear that the buddies may create complications during modification of the directory entries. Factors like directory entry level, the axis along which partition has taken place, and presence of buddies need to be taken into consideration during transformation of directory entries.

Modification Of Directory Entries

The steps involved in modification of directory entries to add an attribute are:

1) Delete the n records which require that an additional attribute to be added, from the existing database
2) Transform the directory entries
3) Insert the n records with new number of attributes to the database with modified attributes.

Some assumptions are necessary for the success of this approach, and are listed below:

- There is a priori knowledge of the maximum number M (or upper bound) of attributes the database will have.

- Each record has enough space to hold the maximum number of attributes.
- All the values of attributes of N+1 through M are set to zero, where N (N < M) is the current dimension of the database.

Steps 1 and 3 deal with the standard record update in a BANG file and do not need an explanation. Step 2 is the heart of the new method, and will be discussed in detail.

Transformation of Directory Entries

As discussed previously, the presence of buddies do not allow for a simple transformation of directory entries. The complications of the buddies can be eliminated by an additional step in the transformation that converts buddies into enclosing regions. Through this step, a simple yet general transformation can be formulated. Given a set of directory entries for N dimensional data, the steps for transformation are:

i) Convert buddies into enclosing regions for the existing N dimension.

ii) Transform the entries obtained from step i, using the transformation of single entries, described below. This gives the entries for N+1 dimensions in the enclosing regions form.

iii) Convert the enclosing regions into buddies, which reflect the addition of the N+1th dimension.

Transformation Of Single Entries Let us consider the transformation, for entries <7,3>, <18,5>, and <81,7> individually. Take the binary representation of the region of the entry, <7,3>

```
x  y  x                    x  z  y  x
1  1  1    ---------->     1  0  1  1
< 7,3 > is transformed to <11, 4>
```

```
x  y  x  y  x              x  z  y  x  z  y  x
1  0  0  1  0   ------>    1  0  0  0  0  1  0
<18, 5> is transformed to  <66,7>
```

```
x  y  x  y  x  y  x            x z x y z y x z y x
1  0  1  0  0  0  1  ------>   1 0 0 1 0 0 0 0 0 1
<81, 7> is transformed to <577,10>
```

During the transformation, a bit '0'(underlined) is inserted at the end the x,y cycle, and it corresponds to the new dimension (attribute) Z. The reason is, all the z values are zero, and every time a partition is made on Z axis, it does not affect the balancing since all the entries still lie in the lower portion after partition. A function (procedure MOD_SUBDIR()) for the transformation of entries is included in the Appendix.

An Example of Transformation of Directory Entries

Every entry in the directory is transformed as explained above, except that we have to account for the existence of the buddy of the entry being transformed. As an illustration, let the contents of a subdirectory be as indicated in Figure 3.5. The operations needed to transform the directory are described below.

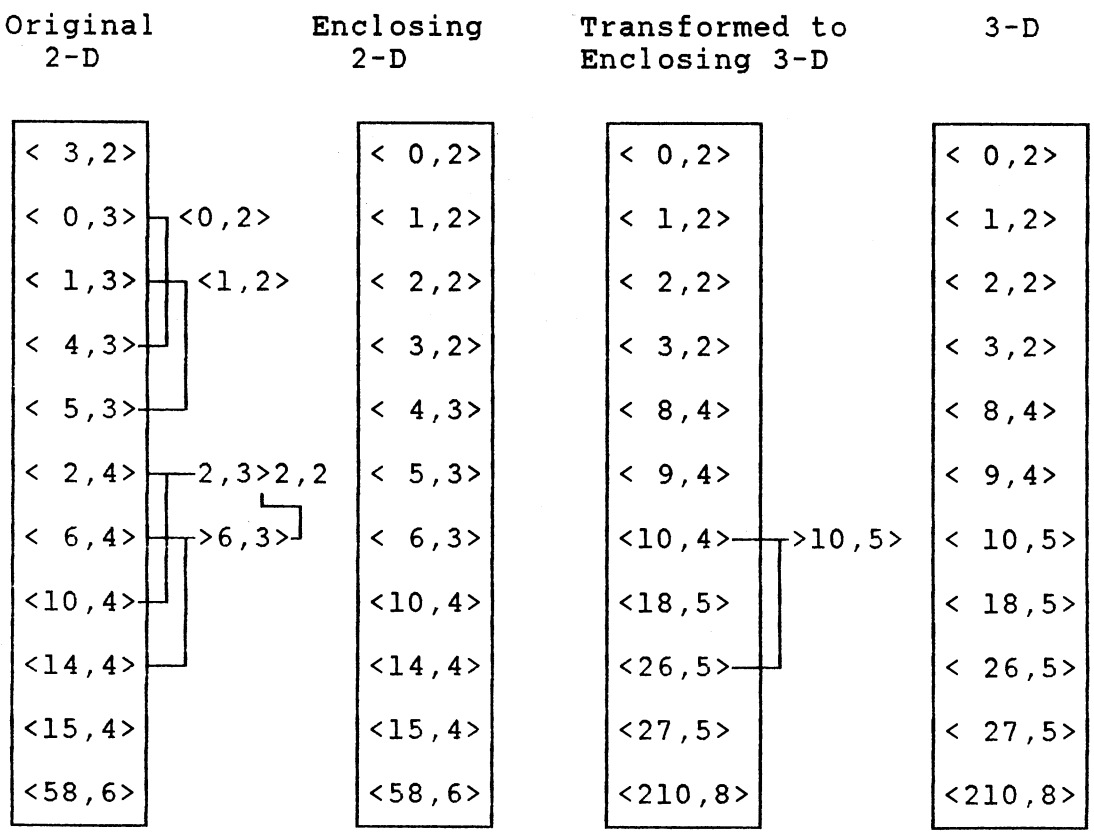


Figure 3.5. Transformation of Directory Entries During the Addition of Attribute

Start from the last entry in the bucket. If its buddy exits, then convert the entry into an enclosing region. If a buddy is not found, then proceed to the next entry. Also, if the entry itself is a higher buddy, go to the next entry. Only the lower buddy gets converted into enclosing region. Process all the entries. In the above example, entries <6,3>, <2,3>, <1,3>, and <0,3> get converted.

At the end of this step, we have buddy free entries in the original dimension. From this point onwards, every entry is transformed individually as explained for individual entries. The transformed entries are the entries of the database with the increase of attribute.

These entries do not contain any buddies. The last step is to convert the enclosing regions into buddies if possible. In the above example, entry <10,4> gets converted into <10,5> because of <26,5>; this defines the areas more specifically.

The final entries are the modified entries. These represent the original data which is N dimensional, modified to N+1 dimensions, and the values corresponding to the N+1th dimension are zero.

Removal Of The Attribute/s

The removal of attributes is also based on the geometrical approach. In this case it is necessary to assume that the values of the attributes that are being

removed are zero.

When any dimension is removed from a data base with H dimensions, the data can be represented in H-1 (L) dimension by suitable modifications to the directory without redistributing the records. As in addition of attributes, the simple approach fails in the case of buddies, and it is necessary to adopt a more sophisticated approach.

Modification of The Directory Entries

Consider an H-dimensional data distribution in a data base; assume that we want to lower the dimension to L. Set all the appropriate coordinate values to zero. Given a set of directory entries for H dimensional data, the steps in modifying the directory to a lower dimensional distribution are as follows:

i. Delete the n records for which attributes needs to be removed.

ii. Modify the directory entries.

iii. Insert the n records with the modified attributes.

Step ii. is again the key part for the removal of an attribute and is discussed in detail below.

Transformation of Directory Entries

The steps for transformation of entries follow the steps in the addition of attributes and are listed below.

i. Convert all the buddies into enclosing regions.

ii. Transform the entries using transformation of single entries described below, this gives the entries for L dimensions.

iii. If possible, convert the enclosing into buddies.

At the end of step iii. we have the L dimensional data.

Transformation of Single Entries Let us consider single entries for a 3-D data, with X, Y, and Z axis.

For an entry <26,5> remove X attribute.

y	**x**	z	y	**x**		y	z	y
1	1	0	1	**0**	------->	1	0	1

Entry <26,5> is transformed into <5,3>

For the same entry <26,5> remove Y attribute

y	x	z	**y**	x		x	z	x
1	1	0	1	0	------->	1	0	0

Entry <26,5> is transformed into <4,3>

When Z attribute is removed,

y	x	**z**	y	x		y	x	y	x
1	1	**0**	1	0	------->	1	1	1	0

Entry <26,5> is transformed into <12,4>

When any attribute is being removed, the corresponding bits are dropped from the original bit string to obtain the modified entry. This is possible because this attribute does not exist any more, and thus during balancing, there is no partitioning on the dimension being removed. A

function (procedure MOD_SUBDIR_DECR()) to remove attributes is presented in the Appendix.

An Example of Transformation of Directory Entries

Figure 3.6 illustrates the directory modification associated with the removal of an attribute from a 3-D data set. The steps are similar to those described in the addition of attributes.

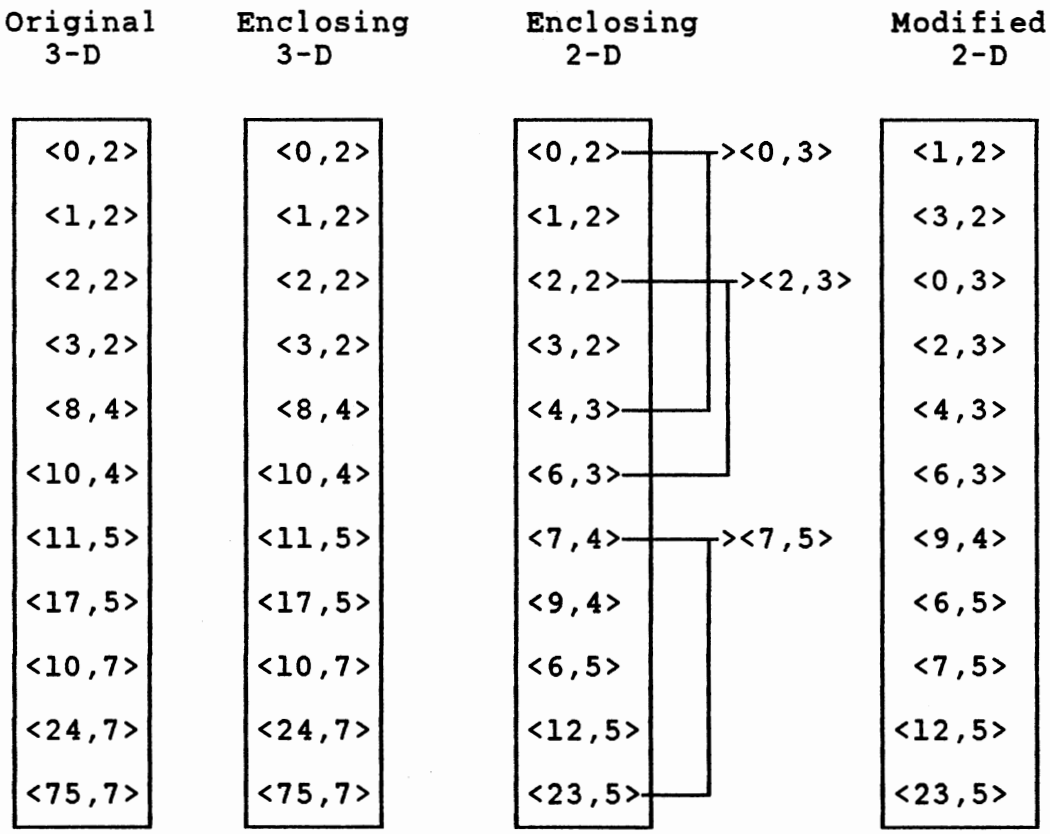


Figure 3.6. Transformation of Directory Entries During the Removal of Attribute

There are no buddies in the original 3-D entries. Consequently the entries remain unchanged. These entries are transformed as single entries into 2-D. At this stage all the regions are in the enclosing form. It is possible to convert some of the entries into buddies. The enclosing entry <0,2>, and entry <4,3> together form buddies, thus entry <0,2> can be converted to <0,3>. Similarly entries <2,2> and <7,4> get converted into <2,3>, and <7,5> respectively. At the end of the modification, the entries represent the original database with 2 attributes.

The 'Directory Modification' approach has been implemented for both addition and removal of attributes, and the results for various data are presented in Chapter IV. It should be noted that the modified entries, do not exactly match the entries obtained from redistribution of records, for that dimension. This is because the directory entries are not unique (as pointed out in Chapter II), and there will not in general be perfect agreement between the two sets of directory entries.

Multiple BANG Files

This approach for the addition and removal of attributes differs from the previous technique by maintaining separate BANG files for each unique combination of attributes .

Multiple files are generated to accommodate changes in the dimension or the combination of attributes in the

database. Addition, and deletion of attributes are described first followed by the organization of the BANG files.

Addition Of Attributes

Let there be a database with 2 attributes say 'a', and 'b', stored in file F1 Figure 3.7a. A third attribute 'c' needs to be added to some of the records 2, 4, and 5.

Records	File F1
1	a1, b1
2	a2, b2
3	a3, b3
4	a4, b4
5	a5, b5
6	a6, b6

(a) Before Addition

File F1	File F2
a1, b1	a2, b2, c2
a3, b3	a4, b4, c4
a6, b6	a5, b5, c5

(b) After Addition

Figure 3.7. Addition of Attribute 'c'

Remove the records 2, 4, and 5 from the existing file F1. Create a new BANG file F2, for three attributes and for the combination 'abc' (if nonexistent), and insert records 2,4, and 5 with the new attribute value into file F2. Figure 3.7b shows the files F1, and F2 after the addition of the third attribute.

All the files are independent of each other. Any

update or query can be performed on individual files as in single BANG file.

Steps To Add Attributes Consider a BANG file with N attributes. If a new N+1th attribute is added to some of the records (say n), the following steps are needed:

1. Delete the n records with N attributes from the existing file.
2. Check if a BANG file with N+1 attributes and the required combination of attributes exists.
   - If it exits, insert the n records with N+1 attributes into the existing N+1 dimensional BANG file.
   - If it does not exist, create a new BANG file, and insert the n records with new N+1 attributes into it.

Deletion Of Attributes

Deletion of the attribute is done along the same lines as addition of attributes. In the above example, if the attribute 'c' is removed from the third record in file F2, this record is removed from F2, and inserted back into F1. Instead of attribute 'c', if attribute 'b' is removed it will have to be put in a new file F3, since the file for combination 'ac' does not exist. Figure 3.8a, and Figure 3.8b show the file F2, before and after removing the attribute 'b' for the third record.

Records	File F2	File F2	File F3
1	a2, b2, c2	a2, b2, c2	a5, c5
2	a4, b4, c4	a4, b4, c4	
3	a5, b5, c5		
	(a) Before Deletion	(b) After Deletion	

Figure 3.8. Before and After Deletion of Attribute 'b' from record #3

Steps to remove an attribute can be summarized as below.

Steps To Remove Attributes Let there be a BANG file H attributes. A certain attribute needs to be removed for some of the records, say n. New number of attributes for these records be L, where L = H - 1 . Steps involved in removing an attribute are given below.

1. Delete the n records with H attributes from the existing file.
2. Check if file with L attributes and required combination exists.
   - If it exists, insert the n records with new attribute values into this file.
   - If it does not, create a new BANG file for the new combination of attributes and insert the n records.

Organization Of Multiple BANG Files

Consider the 'Traditional' BANG file, where all the

records with different combination are stored in one big file F0, it would look as shown in Figure 3.9a. The 'Multiple BANG' files, corresponding to the same data is shown in Figure 3.9b.

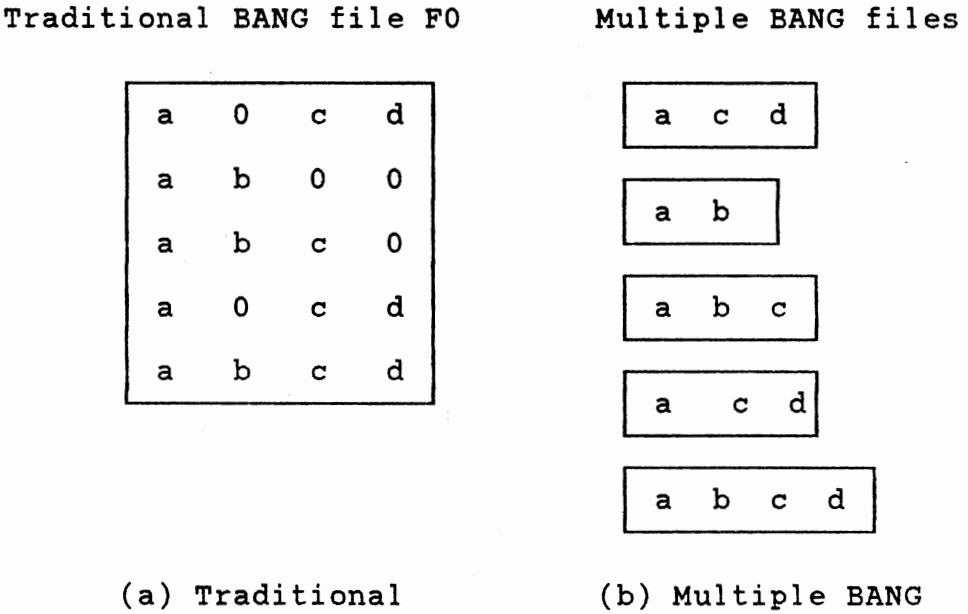


Figure 3.9. Record Distribution in 'Traditional', and 'Multiple BANG' Files

In the above data all the records do not have all 4 attributes. Instead of maintaining one large BANG file with four attributes, the records can be distributed into different BANG files according to the number and combinations of the attributes. Although 'ab' and 'ac' have the same number of attributes, the combination is different and thus are stored in separate files.

Each BANG file contains a unique set of attributes. As the modifications (insertion/deletion) of attributes increase, the combination of attributes also increase, resulting in a proliferation of files ($2^n - 1$, in the worst case, where n is the number of attributes). In the above example, there are 4 attributes, taking all combinations into consideration, there can be 15 files. However, in practical situations, the number will be considerably lower, because all the combinations may not be present.

For any operation, addition/deletion of records, or query it is required to search the BANG file associated with that set of attributes. The operation is performed after locating the required file(s). In case of partial queries, it may be necessary to search more than one file; thus it is essential that we organize the collection of BANG files in such a way that the search is reasonably efficient.

There are many possibilities; one is to store the BANG filename as the nodes of the binary search tree. This structure is quite efficient for an exact query; for a partial query, more than one BANG file may need to be searched. In the worst case, when all the files are to be searched, this will cause a search of the entire tree.

Linked lists (with BANG filename as its elements), are not as efficient as trees for exact queries. But it is easy to update the lists which becomes necessary due to the

addition/removal of attributes. To date there is no best structure available to process range/partial query efficiently. The author proposes the following structure to organize the BANG files.

Let there be 4 attributes ('a', 'b', 'c', 'd') in a database. Figure 3.10 below illustrates all the possible BANG files and organization of these files using a linked list data structure.

An array with the size of the number of attributes is maintained. Each element of the array is associated with a list. Each element in a list is BANG a filename, and the filenames are arranged in alphabetical order.

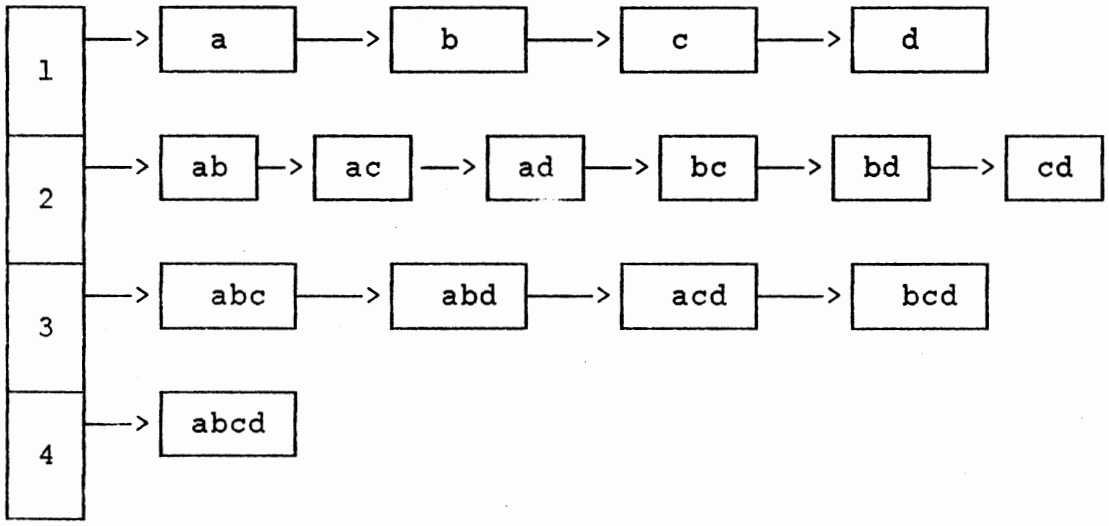


Figure 3.10. Organization of Multiple BANG Files

Note : symbol [ab] represents a BANG file with attributes 'a', and 'b'.

For an exact query of attributes 'abcd', only the list associated with four attributes is searched. In case of a query involving attributes 'a' and 'd', lists of 2, 3, and 4 attribute need to be searched. In this particular case, it becomes a range query for file 'ad' and partial for files 'abd', 'acd', 'bcd', and 'abcd'. Once the files are found, operations on each file are handled as in the case of a single BANG file.

During addition and removal of attribute, the major disk accesses are due to the updates of records, with changes in number of attributes. There is no need to rewrite the entire database. Since there are separate files for each unique set of attributes, there is no need to deal with maximum attributes for every operation, as in the 'Traditional', and 'Directory Modification' techniques.

CHAPTER IV

RESULTS AND DISCUSSIONS

Two new techniques, 'Directory modification', and 'Multiple BANG' files were developed and tested for dynamic modification (addition, and removal) of attributes. Comparison were made with the 'Traditional' BANG file. Results, discussion, and simulation specifications for each method are provided separately. Below are some of the common specifications.

The 'C' language is used for simulation programming on XELOS, which is a Perkin-Elmer 3230 licenced product derived from UNIX SYSTEM V, Release 2.0. Uniform random distributions are used to generate test files. Performance evaluation is based on the number of disk accesses for each of the two proposed techniques 'Directory Modification', and 'Multiple BANG' Files, compared with the 'Traditional' BANG file method.

Results and Discussions for 'Directory Modification'

Evaluation of this technique is based on the number of disk accesses for addition and removal of attributes as compared to the number of disk accesses with the

'Traditional' BANG file method.

The parameters for this BANG file structures are as follows.

Data bucket capacity 64

Block size 2048 bytes

Directory bucket capacity 200

Test file size :

Test File I (n = 1,000, 2,000, and 5,000)

Test File II (n = 1,000, 2,000, and 5,000)

Percent of data, to which an attribute was added :

10%, 25%, 40%, and 50%

Addition of attribute (2-D ---> 3-D)

Start with a 2 dimensional data and modify into 3 dimensional data (Using the 'Directory Modification' technique explained in Chapter III). The algorithm, TRANS_ENTRY() (Appendix) is set to increment one attribute at a time, but if necessary, can be upgraded to handle more than one increment at a time.

Removal of attribute (3-D ---> 2-D)

Start with a three dimensional data, modify it into two dimensional data (using the 'Directory Modification' technique). The algorithm BACK_TRANS() (Appendix) reduces one attribute at a time, however it can be changed to remove more than one attribute at a time.

The 'Directory Modification' technique was used on 2 sets of data, with data sizes of 1,000, 2,000, and 5,000

records in each set. The two sets of data represent different realizations of random numbers and are used to estimate the variability of the results. For each data size, disk accesses, data occupancy, and directory occupancy were computed for addition/deletion of attributes for 10, 25, 40, and 50% of the total number of records. The results for the addition are provided first in Tables VI through XI, followed by the results for the deletion in Tables XII through XVII.

The results of the disk accesses for addition of attributes corresponding to the first and second set are shown on Figures 4.1 and 4.2. They have been normalized by the data accesses for recreation of the BANG file from scratch. The results for deletion are shown in Figures 4.3 and 4.4.

The normalized disk accesses show consistent behavior during addition and deletion of attributes, for both data sets and also between the three data sizes. The relationship can be well approximated by a line through the origin with a slope of 2, especially when the number of records modified is low; consequently, it is seen that the 'Directory Modification' technique is superior to regeneration of the BANG file for addition/deletion of attributes for half the records of the data set.

TABLE VI

ADDITION OF ATTRIBUTE TEST FILE I (N = 1,000)

ADDITION OF ATTRIBUTE FOR 10% OF DATA			
STATISTICS	DATA OCCUPANCY	DIR OCCUPANCY	DISK ACCESSES
TRADITIONAL	45.45	22	4,067
TECHNIQUE 1	45.45	22	806

ADDITION OF ATTRIBUTE FOR 25% OF DATA			
STATISTICS	DATA OCCUPANCY	DIR BUCKETS	DISK ACCESSES
TRADITIONAL	45.45	22	4,067
TECHNIQUE 1	40.00	25	2,018

ADDITION OF ATTRIBUTE FOR 40% OF DATA			
STATISTICS	DATA OCCUPANCY	DIR OCCUPANCY	DISK ACCESSES
TRADITIONAL	50.20	20	4,061
TECHNIQUE 1	43.47	23	3,216

ADDITION OF ATTRIBUTE FOR 50% OF DATA			
STATISTICS	DATA OCCUPANCY	DIR OCCUPANCY	DISK ACCESSES
TRADITIONAL	47.61	21	4,064
TECHNIQUE 1	43.47	23	4,028

TABLE VII

ADDITION OF ATTRIBUTE TEST FILE I (N = 2,000)

ADDITION OF ATTRIBUTE FOR 10% OF DATA			
STATISTICS	DATA OCCUPANCY	DIR OCCUPANCY	DISK ACCESSES
TRADITIONAL	47.61	42	8,127
TECHNIQUE 1	43.47	46	1,604

ADDITION OF ATTRIBUTE FOR 25% OF DATA			
STATISTICS	DATA OCCUPANCY	DIR BUCKETS	DISK ACCESSES
TRADITIONAL	52.63	38	8,115
TECHNIQUE 1	40.00	50	4,022

ADDITION OF ATTRIBUTE FOR 40% OF DATA			
STATISTICS	DATA OCCUPANCY	DIR OCCUPANCY	DISK ACCESSES
TRADITIONAL	48.78	41	8,124
TECHNIQUE 1	45.45	44	6,436

ADDITION OF ATTRIBUTE FOR 50% OF DATA			
STATISTICS	DATA OCCUPANCY	DIR OCCUPANCY	DISK ACCESSES
TRADITIONAL	43.47	46	8,139
TECHNIQUE 1	43.47	46	8,056

TABLE VIII

ADDITION OF ATTRIBUTE TEST FILE I (N = 5,000)

ADDITION OF ATTRIBUTE FOR 10% OF DATA			
STATISTICS	DATA OCCUPANCY	DIR OCCUPANCY	DISK ACCESSES
TRADITIONAL	38.75	129	20,388
TECHNIQUE 1	37.59	133	4,020

ADDITION OF ATTRIBUTE FOR 25% OF DATA			
STATISTICS	DATA OCCUPANCY	DIR BUCKETS	DISK ACCESSES
TRADITIONAL	39.68	126	20,379
TECHNIQUE 1	34.72	144	10,056

ADDITION OF ATTRIBUTE FOR 40% OF DATA			
STATISTICS	DATA OCCUPANCY	DIR OCCUPANCY	DISK ACCESSES
TRADITIONAL	45.87	109	20,328
TECHNIQUE 1	35.97	139	16,126

ADDITION OF ATTRIBUTE FOR 50% OF DATA			
STATISTICS	DATA OCCUPANCY	DIR OCCUPANCY	DISK ACCESSES
TRADITIONAL	44.44	113	20,340
TECHNIQUE 1	40.00	125	20,186

TABLE IX

ADDITION OF ATTRIBUTE TEST FILE II (N = 1,000)

ADDITION OF ATTRIBUTE FOR 10% OF DATA			
STATISTICS	DATA OCCUPANCY	DIR OCCUPANCY	DISK ACCESSES
TRADITIONAL	43.47	23	4,070
TECHNIQUE 1	40.00	25	812

ADDITION OF ATTRIBUTE FOR 25% OF DATA			
STATISTICS	DATA OCCUPANCY	DIR BUCKETS	DISK ACCESSES
TRADITIONAL	45.45	22	4,067
TECHNIQUE 1	40.00	25	2,010

ADDITION OF ATTRIBUTE FOR 40% OF DATA			
STATISTICS	DATA OCCUPANCY	DIR OCCUPANCY	DISK ACCESSES
TRADITIONAL	50.00	20	4,061
TECHNIQUE 1	41.66	24	3,224

ADDITION OF ATTRIBUTE FOR 50% OF DATA			
STATISTICS	DATA OCCUPANCY	DIR OCCUPANCY	DISK ACCESSES
TRADITIONAL	47.61	21	4,064
TECHNIQUE 1	45.45	22	4,026

TABLE X

ADDITION OF ATTRIBUTE TEST FILE II (N = 2,000)

ADDITION OF ATTRIBUTE FOR 10% OF DATA			
STATISTICS	DATA OCCUPANCY	DIR OCCUPANCY	DISK ACCESSES
TRADITIONAL	47.61	42	8,127
TECHNIQUE 1	44.44	45	1,606

ADDITION OF ATTRIBUTE FOR 25% OF DATA			
STATISTICS	DATA OCCUPANCY	DIR BUCKETS	DISK ACCESSES
TRADITIONAL	50.00	40	8,121
TECHNIQUE 1	40.81	49	4,020

ADDITION OF ATTRIBUTE FOR 40% OF DATA			
STATISTICS	DATA OCCUPANCY	DIR OCCUPANCY	DISK ACCESSES
TRADITIONAL	44.44	45	8,136
TECHNIQUE 1	42.55	47	6,408

ADDITION OF ATTRIBUTE FOR 50% OF DATA			
STATISTICS	DATA OCCUPANCY	DIR OCCUPANCY	DISK ACCESSES
TRADITIONAL	43.47	46	8,139
TECHNIQUE 1	43.47	46	8,052

TABLE XI

ADDITION OF ATTRIBUTE TEST FILE II (N = 5,000)

ADDITION OF ATTRIBUTE FOR 10% OF DATA			
STATISTICS	DATA OCCUPANCY	DIR OCCUPANCY	DISK ACCESSES
TRADITIONAL	39.06	128	20,385
TECHNIQUE 1	37.59	133	4,024

ADDITION OF ATTRIBUTE FOR 25% OF DATA			
STATISTICS	DATA OCCUPANCY	DIR BUCKETS	DISK ACCESSES
TRADITIONAL	41.32	121	20,364
TECHNIQUE 1	34.72	144	10,060

ADDITION OF ATTRIBUTE FOR 40% OF DATA			
STATISTICS	DATA OCCUPANCY	DIR OCCUPANCY	DISK ACCESSES
TRADITIONAL	46.29	108	20,325
TECHNIQUE 1	35.71	140	16,116

ADDITION OF ATTRIBUTE FOR 50% OF DATA			
STATISTICS	DATA OCCUPANCY	DIR OCCUPANCY	DISK ACCESSES
TRADITIONAL	44.24	113	20,340
TECHNIQUE 1	40.00	125	20,184

TABLE XII

REMOVAL OF ATTRIBUTE
TEST FILE I (N = 1000)

METHOD	DIRECTORY MODIFICATION				TRADITIONAL
	REMOVAL OF ATTRIBUTE FOR				
	10%	25%	40%	50%	
DATA OCCUPANCY	41.66	47.61	47.61	47.61	47.61
DIR OCCUPANCY	24	21	21	21	21
DISK ACCESSES	810	2,014	3,219	4,020	4,060

TABLE XIII

REMOVAL OF ATTRIBUTE
TEST FILE I (N = 2000)

METHOD	DIRECTORY MODIFICATION				TRADITIONAL
	REMOVAL OF ATTRIBUTE FOR				
	10%	25%	40%	50%	
DATA OCCUPANCY	44.44	44.44	44.44	44.44	44.44
DIR OCCUPANCY	45	45	45	45	45
DISK ACCESSES	1,626	4,040	6,448	8,052	8,136

TABLE XIV

REMOVAL OF ATTRIBUTE
TEST FILE I (N = 5000)

METHOD	DIRECTORY MODIFICATION				TRADITIONAL
	REMOVAL OF ATTRIBUTE FOR				
	10%	25%	40%	50%	
DATA OCCUPANCY	39.37	39.37	39.37	39.37	39.37
DIR OCCUPANCY	127	127	127	127	127
DISK ACCESSES	4,016	10,087	16,202	20,226	20,382

TABLE XV

REMOVAL OF ATTRIBUTE
TEST FILE II (N = 1,000)

METHODS	DIRECTORY MODIFICATION				TRADITIONAL
	REMOVAL OF ATTRIBUTE FOR				
	10%	25%	40%	50%	
DATA OCCUPANCY	43.47	43.47	43.47	43.47	43.47
DIR OCCUPANCY	23	23	23	23	23
DISK ACCESSES	806	2,006	3,125	4,038	4,070

TABLE XVI

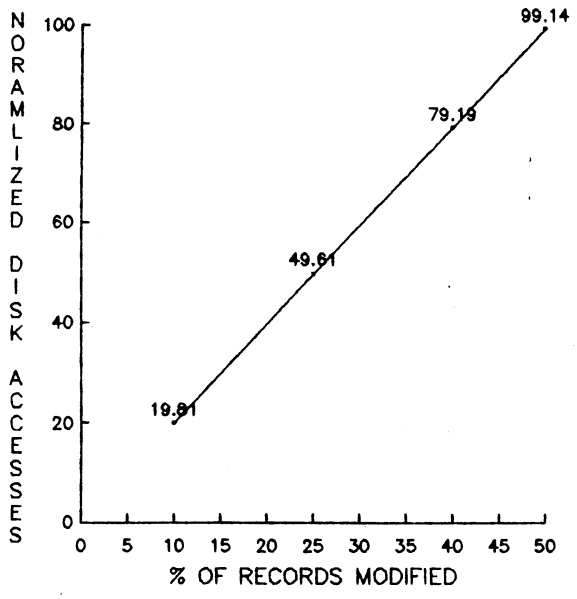
REMOVAL OF ATTRIBUTE
TEST FILE II (N = 2,000)

METHODS	DIRECTORY MODIFICATION				TRADITIONAL
	REMOVAL OF ATTRIBUTE FOR				
	10%	25%	40%	50%	
DATA OCCUPANCY	41.66	41.66	41.66	41.66	42.55
DIR OCCUPANCY	48	48	48	48	47
DISK ACCESSES	1,611	4,036	6,454	8,076	8,142

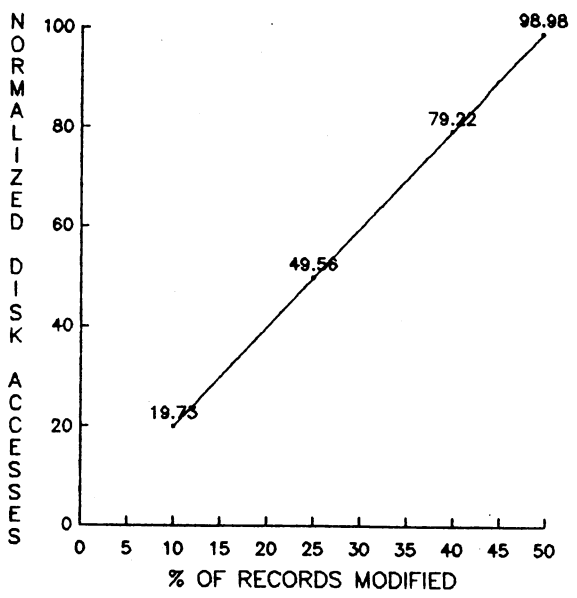
TABLE XVII

REMOVAL OF ATTRIBUTE
TEST FILE II (N = 5,000)

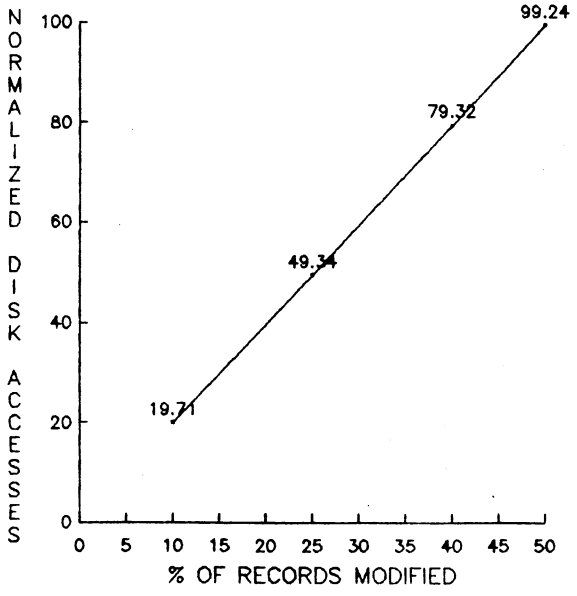
METHOD	DIRECTORY MODIFICATION				TRADITIONAL
	REMOVAL OF ATTRIBUTE FOR				
	10%	25%	40%	50%	
DATA OCCUPANCY	49.01	49.01	49.01	49.01	49.01
DIR OCCUPANCY	102	102	102	102	102
DISK ACCESSES	4,020	10,050	16,082	20,122	20,308



(a) Test File I (n = 1,000)

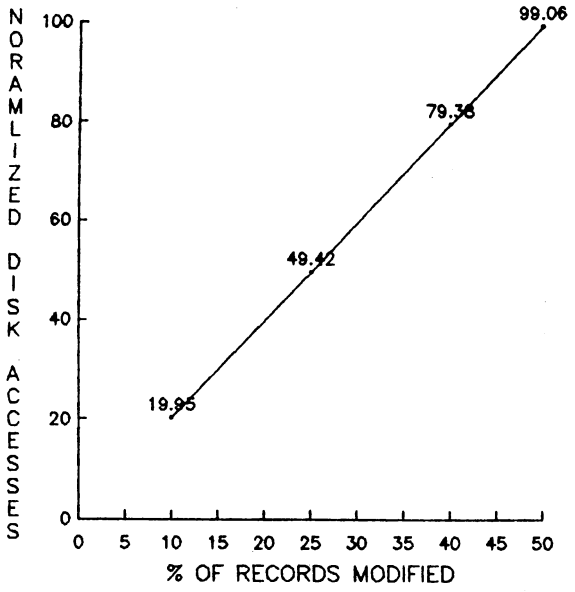


(b) Test File I (n = 2,000)

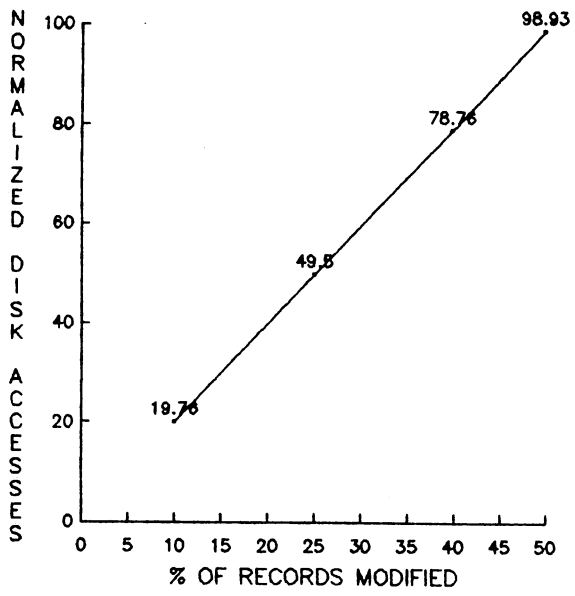


(c) Test File I (n = 5,000)

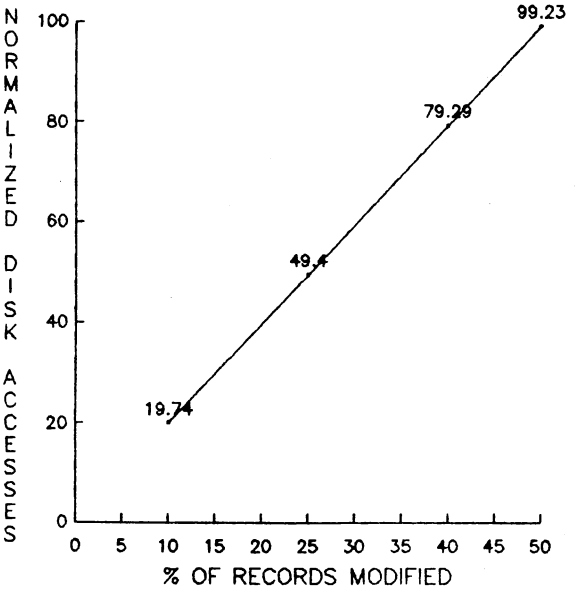
Figure 4.1. Dependence of Normalized Disk Accesses on Records Modified - Attribute Addition



(a) Test File II (n = 1,000)

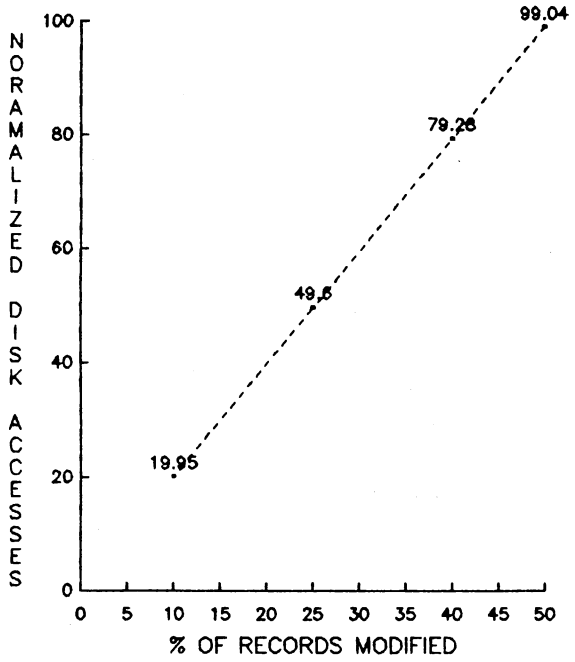


(b) Test File II (n = 2,000)

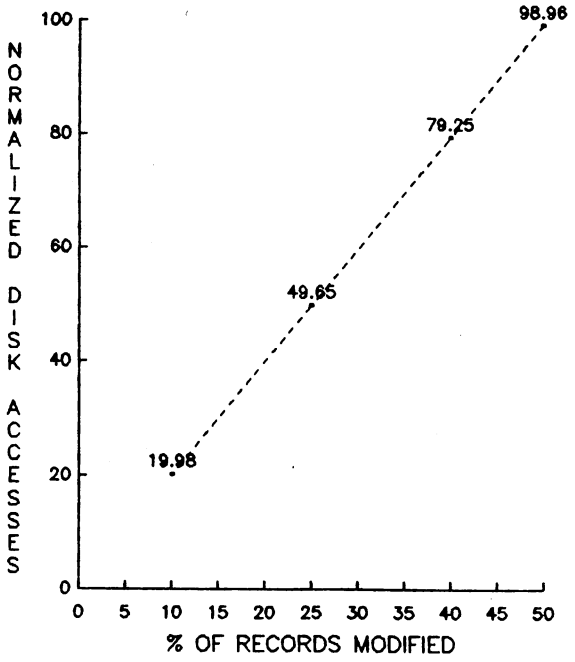


(c) Test File II (n = 5,000)

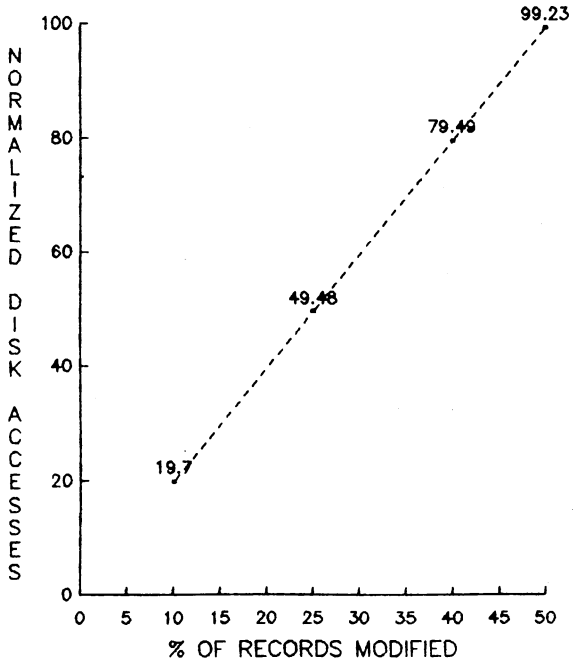
Figure 4.2. Dependence of Normalized Disk Accesses on Records Modified - Attribute Addition



(a) Test File I (n = 1,000)



(b) Test File I (n = 2,000)



(c) Test File I (n = 5,000)

Figure 4.3. Dependence of Normalized Disk Accesses on Records Modified - Attribute Deletion

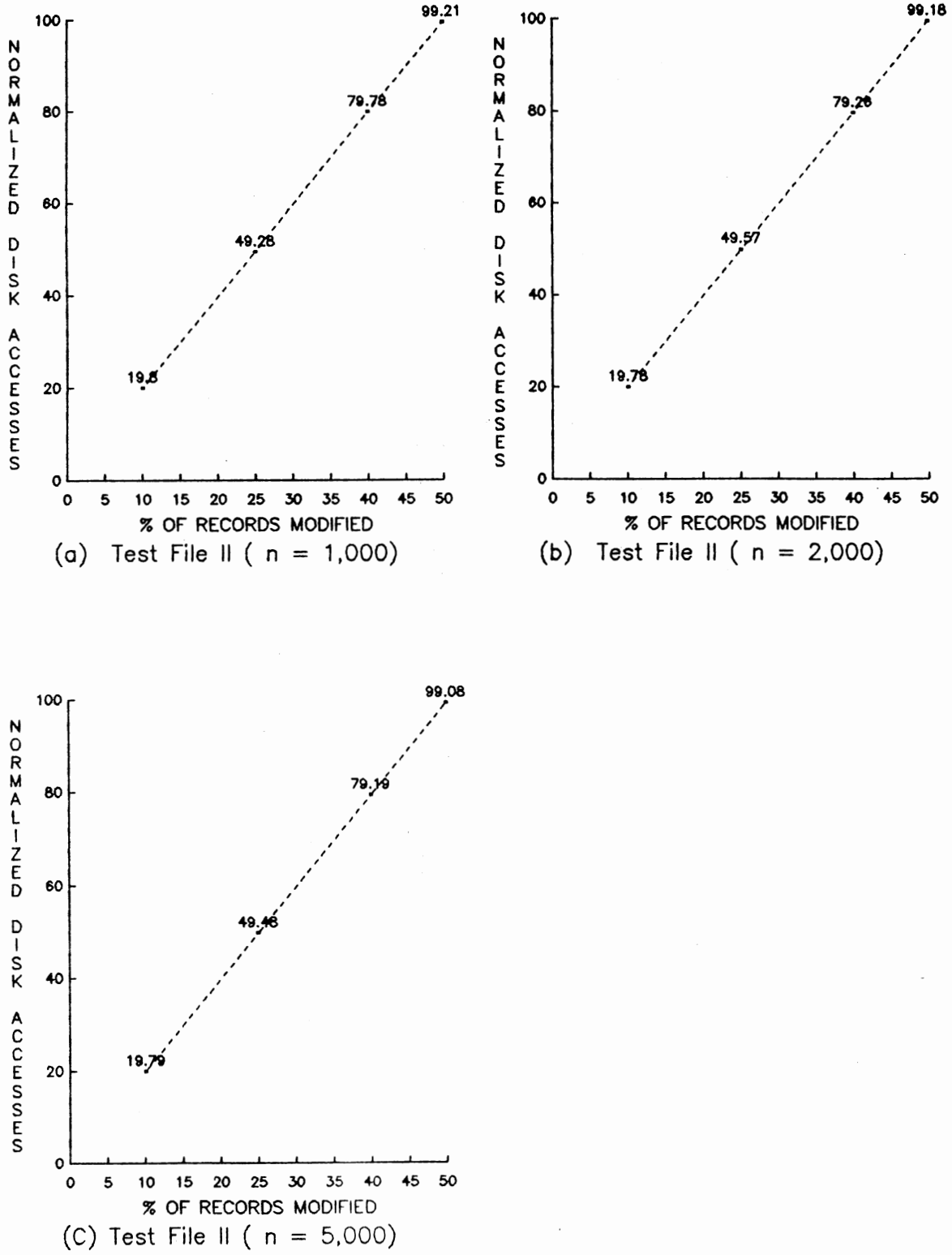


Figure 4.4. Dependence of Normalized Disk Accesses on Records Modified - Attribute Deletion

The linear relationship reflects the two operations of deletion and addition associated with 'Directory Modification'. Since generation of a BANG file involves only the operation of addition, the slope of the line is two. For practical problems, a small number of additional data accesses are required to handle directory splitting and merging. Attribute addition, in general, results in slightly lower data occupancy than with the BANG file regenerated. The data occupancy appears to be lower for about 25 to 40% of the records. Attribute deletion however, does not show any decrease in the data occupancy.

The explanation for this is as follows: attribute addition is accomplished by deleting a record followed by inserting the new record with the additional attribute. The deletion of records causes a decrease in the data occupancy. The addition of an attribute is done in a new dimension and consequently needs additional directories leading to lower data occupancy. Deletion of attribute is also accomplished by deletion of record. However, this redistributes the data into a lower dimension and is accomplished by a reduction in directory buckets. Addition of the record is done later in a lower dimension; this compensates for the loss of data occupancy.

Results And Discussions for 'Multiple BANG' Files

The second technique, the 'Multiple BANG' files technique, stores the data in multiple files. A separate BANG file is created for each unique combination of attributes.

There are two important considerations in evaluating the Multiple BANG file approach; one deals with the disk accesses connected with the creation of the file, and the second with the accesses connected with the queries. The disk accesses associated with the addition, and deletion of records are well understood: namely four disk accesses for each operation. Consequently, testing of the method was restricted exclusively to disk accesses resulting from queries.

Queries may be divided into three types: exact queries, range queries, and partial queries. In exact queries, the values of all the attributes are specified, and finding the record consists of searching the bucket in which the record is stored. Exact queries require two disk accesses for both 'Traditional' and 'Multiple BANG' files; thus there is no need to perform any tests, since the results are known in advance.

Range queries specify the range of all attributes. In partial queries, the range is specified for one or more attributes but not for all the attributes. All allowable values are accepted for attributes whose range are not

specified.

This section deals with the comparison of disk accesses for range and partial queries. The following common set of BANG file parameters were used in testing all queries:

Block size	2048 bytes
Data bucket capacity	32
Directory bucket capacity	8

Results And Discussions For Range Queries

Comparison of disk accesses for the 'Traditional', and the 'Multiple BANG' files were performed using the following data examples.

Dimension	3
Data size	1,000, 2,000, and 5,000
Range sizes	10%, 20%, 30%, 40%, and 50%
Third attribute specified for	25%, and 50% of total records

The three attribute (3-D) simulation was carried out for three data sizes. For each data size, the third attribute was specified for 25%, and 50% of the total records. Consider an example of 2,000 records with three attributes 'a', 'b', and 'c'. A value for 'c' may be specified for either 25% (500 records), or 50% (1000 records) for the data; the rest of the records are assigned

a zero value for 'c' attribute.

The range query was performed on three attributes, and the range size was varied from 10 to 50% in increments of 10%. Values of 0, and 50% were selected and the results shown in Tables XVIII through XXIII represent average values.

The results of the disk accesses for each of three data sizes are consistent. Consequently, the results can be discussed for one data size, namely, 5,000 records. The data distribution is uniform, and the number of records found in any range are given by the product of the range in the three dimensions, the fraction of the records having those attributes, and the total number of records.

For range query of 30%, with 5,000 records, and 50% with a third attribute, TABLE XXIII the number of records found should be

$$(.3)*(.3)*(.3)*(.5)*(5000) = 67.5$$

The number observed in the table is 70.5, which is within 5% of the expected value.

TABLE XVIII

RANGE RETRIEVAL (N = 1,000)
25% DATA WITH 3rd ATTRIBUTE

Range size	10%	20%	30%	40%	50%
Avg Records found	0.0	2.0	7.5	19.0	34.5
Avg Disk Accesses for Traditional BANG file	3.0	3.0	6.0	7.0	16.0
Avg Disk Accesses for Multiple BANG files	2.0	2.0	4.0	4.0	8.0

TABLE XIX

RANGE RETRIEVAL (N = 1,000)
50% DATA WITH 3rd ATTRIBUTE

Range size	10%	20%	30%	40%	50%
Avg Records found	1.5	6.0	16.5	39.5	68.0
Avg Disk Accessed for Traditional BANG file	2.0	3.0	8.0	8.0	20.0
Avg Disk Accesses for Multiple BANG files	2.0	2.0	5.0	5.0	12.0

TABLE XX

RANGE RETRIEVAL (N = 2,000)
25% DATA WITH 3rd ATTRIBUTE

Range size	10%	20%	30%	40%	50%
Avg Records Found	1.5	6.0	16.5	39.5	68.0
Avg Disk Accesses for Traditional BANG file	3.0	4.0	10.0	13.0	26.0
Avg Disk Accesses for Multiple BANG files	2.0	2.0	5.0	5.0	12.0

TABLE XXI

RANGE RETRIEVAL (N = 2,000)
50% DATA WITH 3rd ATTRIBUTE

Range size	10%	20%	30%	40%	50%
Avg Records Found	2.5	12.0	33.0	79.5	137.0
Avg Disk Accesses for Traditional BANG file	2.0	4.0	13.0	15.0	31.0
Avg Disk Accesses for Multiple BANG files	2.0	2.0	8.0	8.0	19.0

TABLE XXII

RANGE RETRIEVAL (N = 5,000)
25% DATA WITH 3rd ATTRIBUTE

Range size	10%	20%	30%	40%	50%
Avg Records Found	3.0	15.0	42.5	99.0	172.5
Avg Disk Accesses for Traditional BANG file	3.0	8.0	21.0	32.0	54.0
Avg Disk Accesses for Multiple BANG files	2.0	2.0	10.0	10.0	22.0

TABLE XXIII

RANGE RETRIEVAL (N = 5,000)
50% DATA WITH 3rd ATTRIBUTE

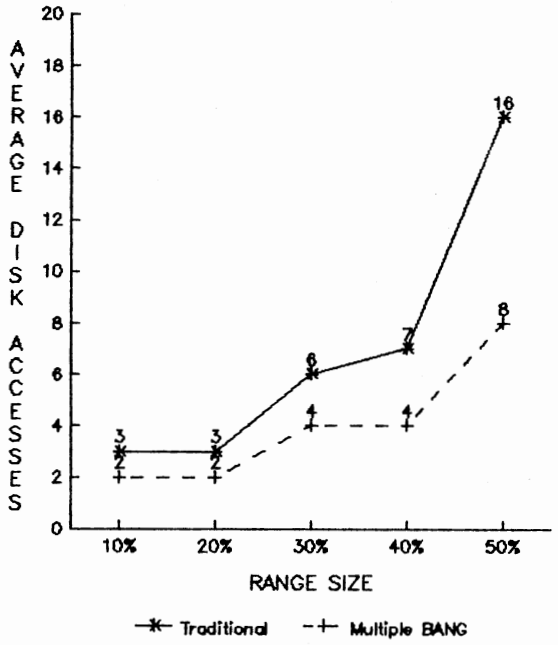
Range size	10%	20%	30%	40%	50%
Avg Records Found	3.5	21	70.5	177.5	317.5
Avg Records Found for Traditional BANG file	4.0	7.0	17.0	32.0	58.0
Avg Records Found for Multiple BANG files	3.0	4.0	14.0	17.0	35.0

Figures 4.5, 4.6, and 4.7 show the consistent reduction in disk accesses for the 'Multiple BANG' files technique over the 'Traditional BANG' file for all range values. As the range increases, the difference gets larger, and the superiority of the 'Multiple BANG' is clearly demonstrated.

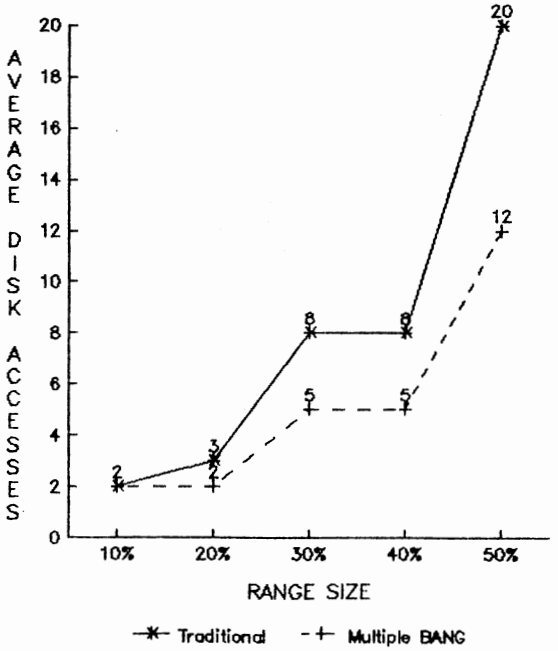
Results may also be analyzed in terms of the percent of data to be searched. As the percentage of records with the third attribute gets lower, the difference gets larger,i.e the results are better for the case where the third attribute is specified for 25% rather than for 50%. The reduced disk accesses come from the searching of fewer records.

Disk accesses show a non-linear dependence on the range. In general disk accesses appear to increase as a power of the range. This pattern is seen for the two cases with 25%, and 50% of the third attribute defined. Additionally a simple doubling from 25% to 50% is not observed. Note that, the bucket capacity of 32 and directory capacity of 8 can not provide statistically significant results, when the number of records accessed is small. This explains the non-smooth behavior observed for the lower range queries.

The 'Traditional' BANG approach does not really scan all the records. The specification of the range eliminates certain buckets within the given region. A comparison of disk accesses for the 'Traditional' BANG method, and

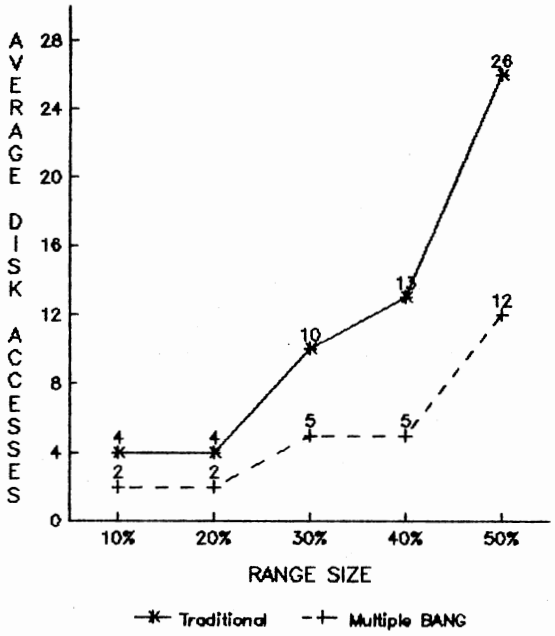


(a) n = 1000 (25% with 3rd attribute)

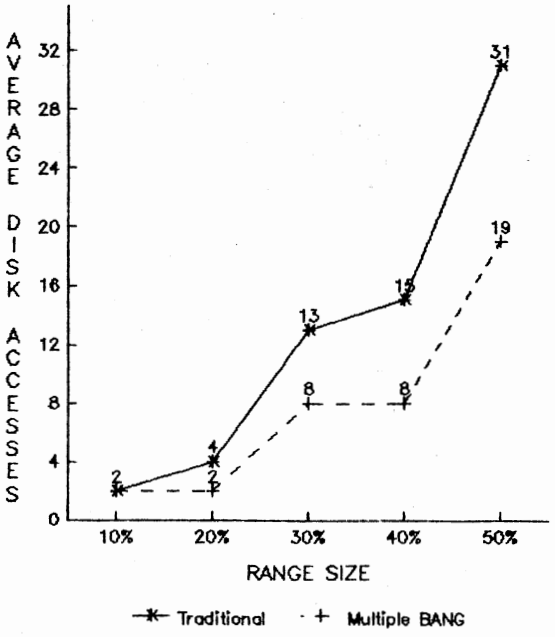


(b) n = 1000 (50% with 3rd attribute)

Figure 4.5. Average Disk Accesses for Range Query (3-D) Traditional Vs Multiple BANG Files

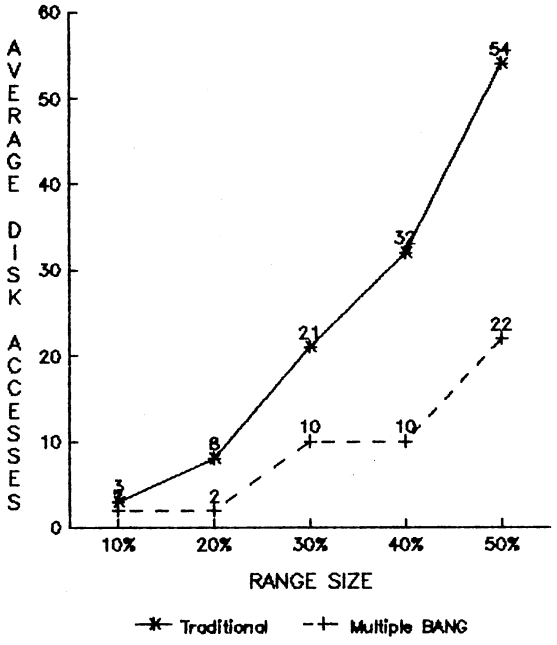


(a) n = 2000 (25% with 3rd attribute)

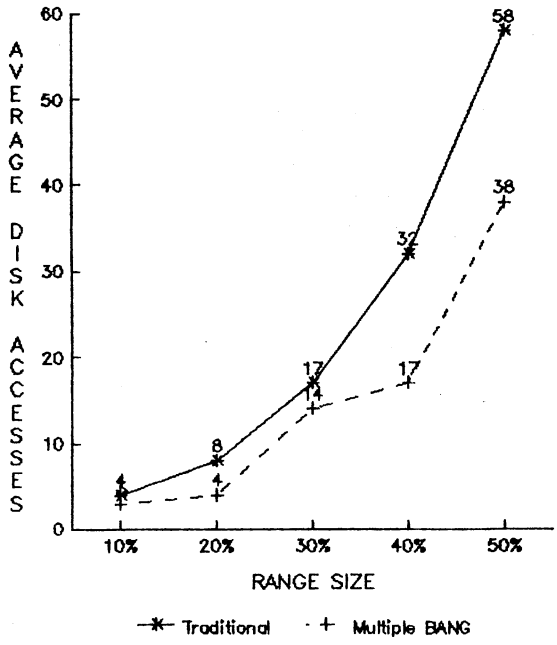


(b) n = 2000 (50% with 3rd attribute)

Figure 4.6. Average Disk Accesses for Range Query (3-D)
Traditional Vs Multiple BANG Files



(a) n = 5000 (25% with 3rd attribute)



(b) n = 5000 (50% with 3rd attribute)

Figure 4.7. Average Disk Accesses for Range Query (3-D)
Traditional Vs Multiple BANG Files

'Multiple BANG' files for the larger ranges, shows a reduction of 60% for the case where 50% of the records have third attribute. The reduction is even more significant, around 45%, for the case where only 25% of the data has the third attribute.

Results And Discussions for Partial Queries

The comparison of disk accesses for partial queries for the 'Traditional', and 'Multiple BANG' files were tested on the following data examples.

Dimension	4
Data size	5,000 records
Range sizes	10%, 30%, and 50%

Four attributes (4-D) were chosen to simulate some of the complexity that can arise in partial queries as the number of attributes is increased. It was also felt that a database with less than 5,000 records would not be statistically acceptable in terms of records encountered.

The database with 4 attributes, 'a', 'b', 'c', and 'd'. Figure 4.8 shows the actual combination of attributes and number of the records distributed in different files, used for partial query in the simulation.

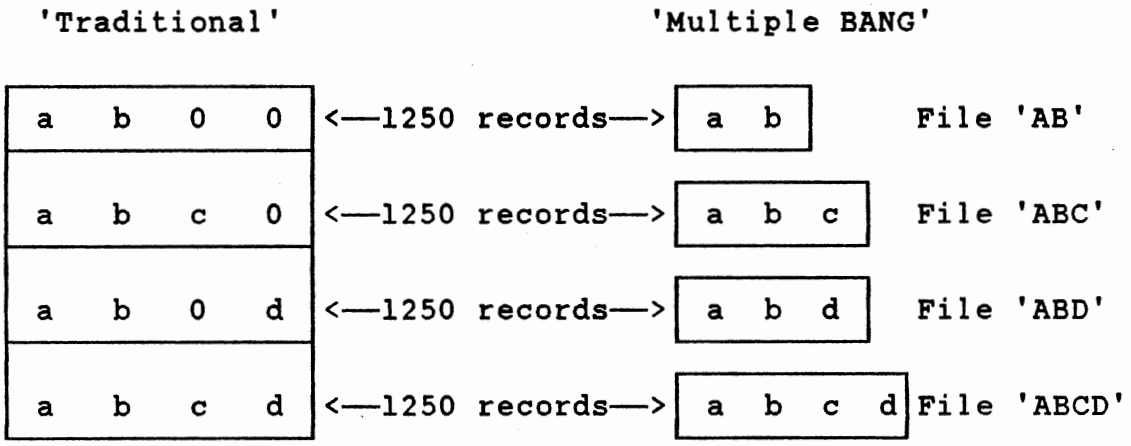


Figure 4.8 'Traditional' and 'Multiple BANG' files used for partial query

In a general four attribute data set, there can be fourteen partial queries, and a range query. The partial queries are 'a', 'b', 'c', 'd', 'ab', 'ac', 'ad', 'bc', 'bd', 'cd', 'abc', 'abd', 'acd', and 'bcd', and the range query is 'abcd'. The data distribution selected here allows attributes 'a', and 'b', and 'c', and 'd' to be interchanged. Consequently only seven types of partial queries namely, 'a', 'c, 'ab', 'ac', 'cd', 'abc', 'acd' and one range query 'abcd' exist.

Partial and range queries were selected for the following combinations 'a', 'ab', 'ac', 'abc', 'bd', 'cd', 'bcd', and 'abcd'. Note that 'bd' provides the 'ac' type of partial query and provides a measure of statistical variability in the results.

A range of 10%, 30%, and 50% is taken for each

combination of attributes. The ranges start at different positions i.e. 0, 25, and 50% for each attribute. The results listed in Tables XXIV through XXXI are average values.

TABLE XXIV

PARTIAL QUERY FOR 'ab' (100% SEARCH)

RANGE SIZE	10%	30%	50%
AVG RECORDS FOUND	51.33	449.00	1249.00
AVG DISK ACCESSES FOR TRADITIONAL	14.66	58.00	114.66
AVG DISK ACCESSES FOR MULT BANG	22.00	69.33	126.66

TABLE XXV

PARTIAL QUERY FOR 'a' (100% SEARCH)

RANGE SIZE	10%	30%	50%
AVG RECORDS FOUND	526.33	1521.00	2525.00
AVG DISK ACCESSES FOR TRADITIONAL	54.00	119.66	168.66
AVG DISK ACCESSES FOR MULT BANG	67.00	133.00	180.66

TABLE XXVI

PARTIAL QUERY FOR 'ac' (50% SEARCH)

RANGE SIZE	10%	30%	50%
AVG RECORDS FOUND	25.66	232.66	637.33
AVG DISK ACCESSES FOR TRADITIONAL	19.33	57.66	102.00
AVG DISK ACCESSES FOR MULT BANG	16.00	44.00	76.33

TABLE XXVII

PARTIAL QUERY FOR 'bd' (50% SEARCH)

RANGE SIZE	10%	30%	50%
AVG RECORDS FOUND	21.33	215.66	599.00
AVG DISK ACCESSES FOR TRADITIONAL	27.33	66.00	112.66
AVG DISK ACCESSES FOR MULT BANG	18.66	46.66	78.00

TABLE XXVIII

PARTIAL QUERY FOR 'abc' (50% SEARCH)

RANGE SIZE	10%	30%	50%
AVG RECORDS FOUND	3.33	71.33	314.33
AVG DISK ACCESSES FOR TRADITIONAL	4.66	28.66	73.33
AVG DISK ACCESSES FOR MULT BANG	6.00	25.33	57.33

TABLE XXIX

PARTIAL QUERY FOR 'cd' (25% SEARCH)

RANGE SIZE	10%	30%	50%
AVG RECORDS FOUND	14.00	116.33	309.33
AVG DISK ACCESSES FOR TRADITIONAL	56.00	87.33	118.66
AVG DISK ACCESSES FOR MULT BANG	19.33	36.66	53.33

TABLE XXX

PARTIAL QUERY FOR 'bcd' (25% SEARCH)

RANGE SIZE	10%	30%	50%
AVG RECORDS FOUND	2.00	35.00	157.33
AVG DISK ACCESSES FOR TRADITIONAL	11.33	42.66	86.66
AVG DISK ACCESSES FOR MULT BANG	3.00	20.66	40.33

TABLE XXXI

RANGE QUERY FOR 'abcd' (25% SEARCH)

RANGE SIZE	10%	30%	50%
AVG RECORDS FOUND	0.00	9.66	76.33
AVG DISK ACCESSES FOR TRADITIONAL	2.66	20.00	63.00
AVG DISK ACCESSES FOR MULT BANG	2.00	11.33	30.00

The observed results appear to depend on two key quantities: the percent of data searched, and the number of attributes used in the query.

A search is said to be 100%, if the search covers the entire file. Similarly 50%, and 25% search indicate the percent of the file to be searched.

Let us consider the results for a two-attribute partial query. Figure 4.9 show the disk accesses for partial queries 'ab, 'bd', 'ac', and 'cd'.

Partial query 'ab' requires a 100% search. It becomes a range query for file 'AB', a partial for 'ABC', 'ABD, and 'ABCD'. The 'Multiple BANG' files technique uses four files, and needs between 8 to 12 additional disk accesses over the single BANG file. The search of three additional BANG files introduces 6 disk accesses (2 per BANG file). There also appears to be a small increase in accesses resulting from multiple accesses of a given 'ab' range in the separate BANG files.

The partial query for 'bd' (and also 'ac') requires a partial search for file 'ABD'('ACD') and 'ABCD', equivalent to a 50% search. The 'Multiple BANG' requires less than 0.7 accesses of the 'Traditional' method. The difference in disk accesses get larger as the range is increased

Partial queries for 'bd' and 'ac', Figure 4.9, show the magnitude of scatter that can arise from the data distribution. The disk accesses show the same trend. The 'Traditional' method appears to require larger number of

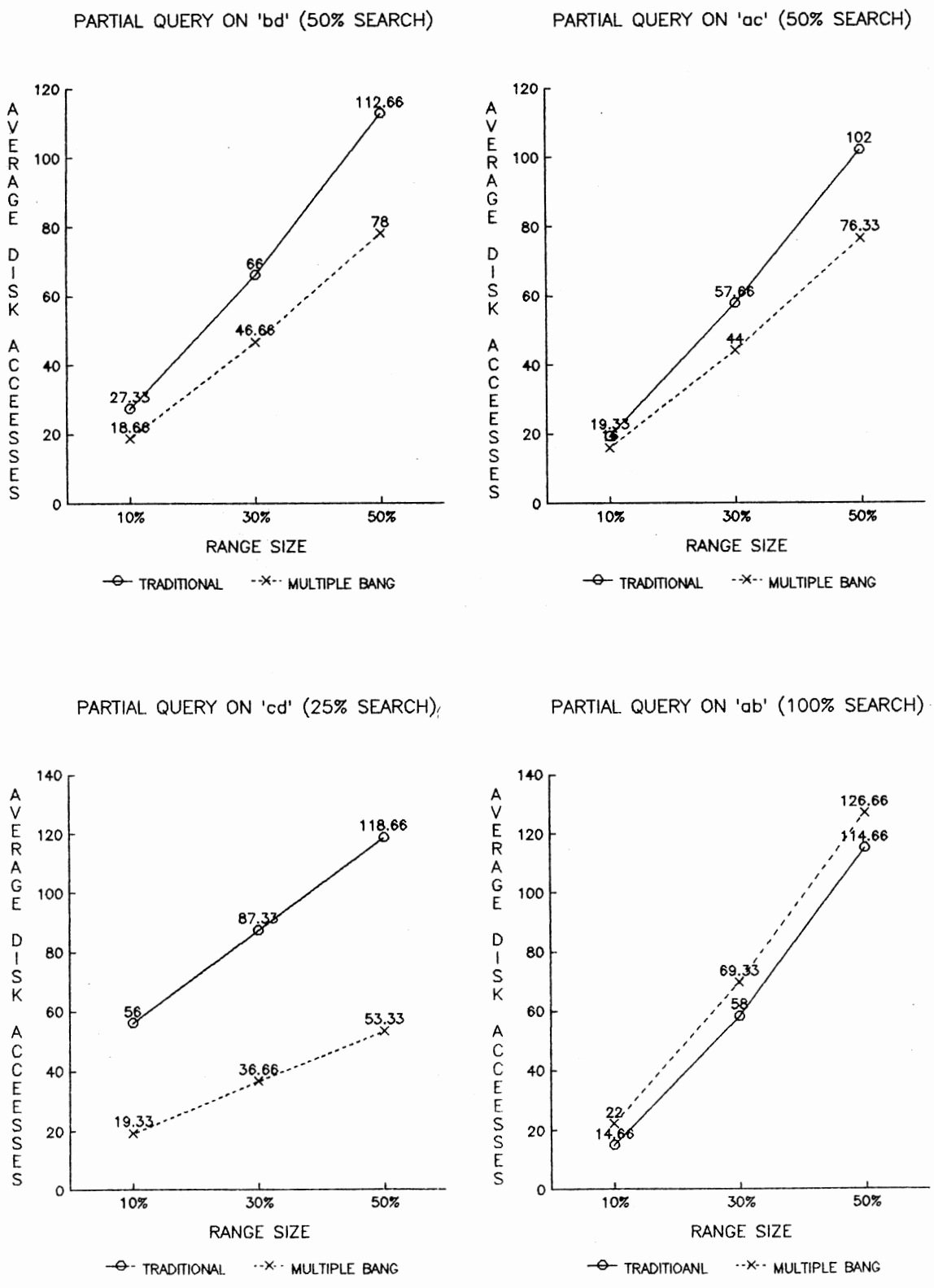


Figure 4.9. Dependence Of Disk Accesses for 3 Search Sizes - 2 Attribute Partial Query

disk accesses for 'bd' as compared to 'ac'.

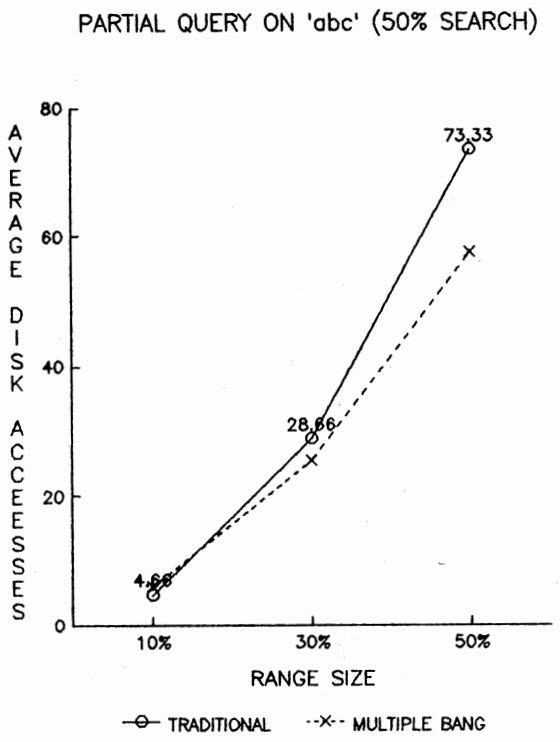
The partial query for 'cd', Figure 4.9, requires a partial search of file 'ABCD' only, a 25% search. The improvement in the disk accesses of the 'Multiple BANG' files are even more dramatic, requiring less than 45% of the accesses of the 'Traditional' approach. Again the absolute difference gets larger as the range is increased.

The partial queries for 3 attributes, 'abc', and 'bcd' Figure 4.10 show similar trends. The 25% search for 'bcd' shows greater reduction in disk accesses over 'Traditional' than does a 50% search for 'abc' over the 'Traditional'.

The effect of the number of attributes on partial/range queries Figure 4.11, is seen by comparing the results of the 25% search, queries for 'cd', 'bcd', and 'abcd'. It is seen that the largest absolute difference in disk accesses between the 'Traditional', and 'Multiple BANG' occur for 'cd' and the smallest for 'abcd'. The volume of the data space that falls within the specified range for a given partial query is related to the range and the number of attributes.

For a given range, this volume decreases as the number of attributes are increased. The ratio of the disk accesses of the 'Multiple BANG' files to the single BANG file for the 50% range increases slightly from 0.45 to 0.46 to 0.48, for 'cd', bcd', and 'abcd' respectively.

The results clearly demonstrate the superiority of the 'Multiple BANG' files over the 'Traditional' method for



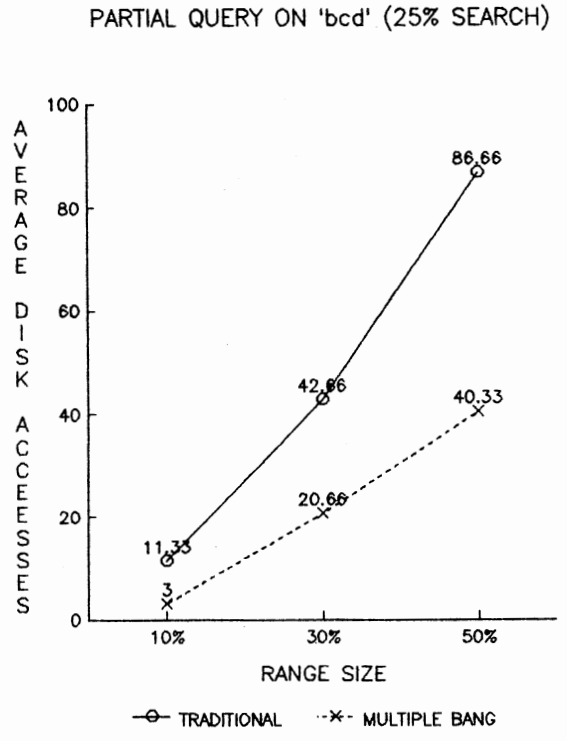


Figure 4.10. Dependence Of Disk Accesses for 2 Search Sizes - 3 Attribute Partial Query

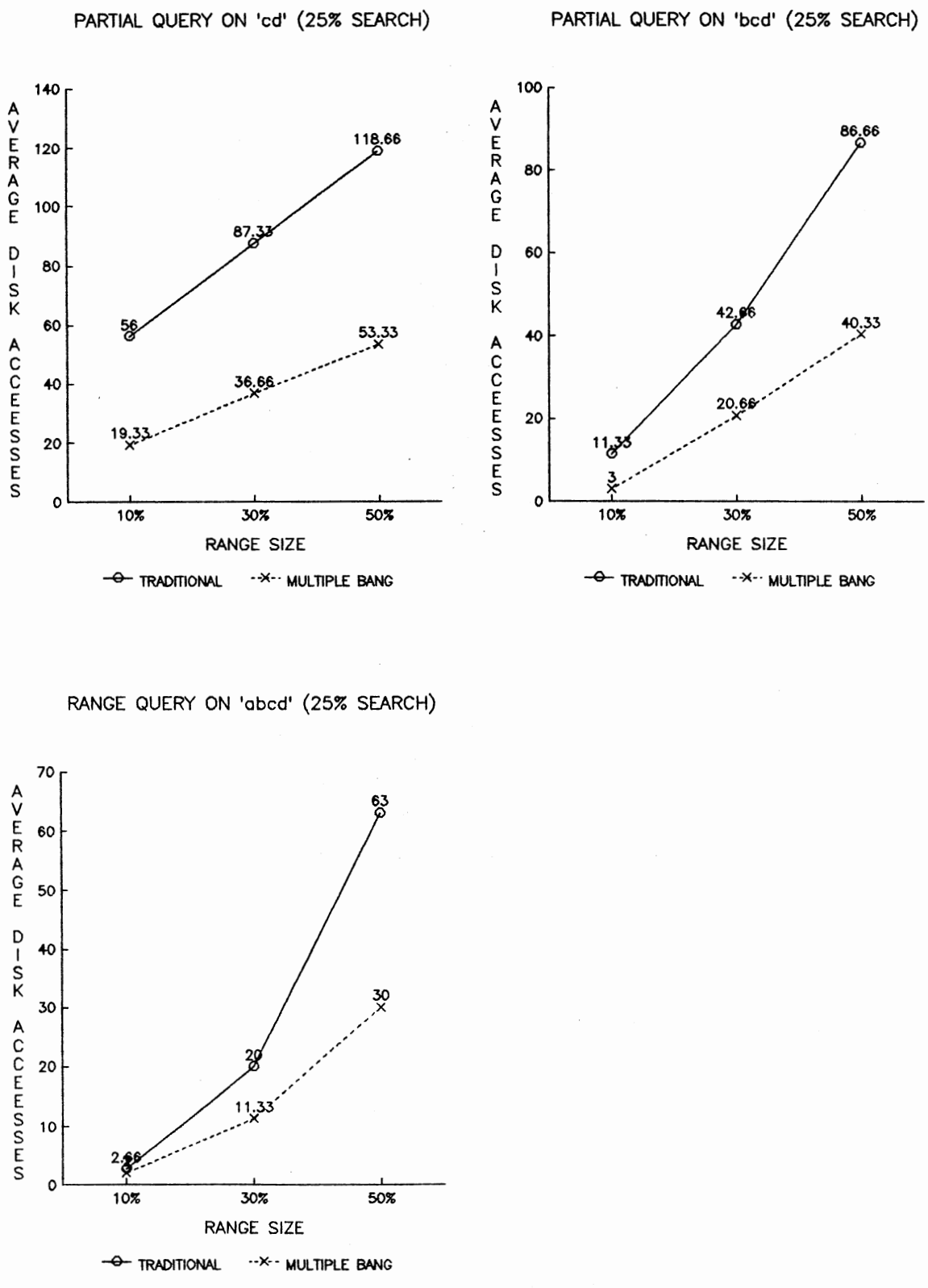


Figure 4.11. Dependence Of Disk Accesses on the Number of Attributes - 25% Search Size

range and partial queries. As the percent of search becomes smaller, dramatic improvements in disk accesses are observed.

CHAPTER V

CONCLUSIONS AND RECOMMENDATIONS

Previous studies have demonstrated the efficiency of the BANG file structure in minimizing accesses to secondary storage. It retains a one-to-one correspondence between a data bucket and a directory entry for any arbitrary distribution of data through a flexible definition of enclosing regions.

The BANG file structure is based on a fixed number of attributes. Updates deal only with the insertion and deletion of records. A number of practical applications however require the addition and removal of attributes to a selected number of records.

Two new techniques, 'Directory Modification' and 'Multiple BANG' files, have been developed and tested to extend the applicability of BANG files. 'Directory modification' preserves the existing data structure of the data and modifies the dimension of the directory to reflect the addition or deletion of attributes. 'Multiple BANG' files generates separate BANG files necessary for each unique combination of attributes. Both these new methods have been developed to deal with changing attributes for a portion of the data, rather than for the whole data set.

The 'Traditional' approach does not use the existing BANG file structure, and has to recreate a BANG file from scratch each time the number of attributes are modified for any record(s) in the data set.

Comparison of both the 'Directory Modification' technique and the 'Multiple BANG Files' technique with the 'Traditional' method, were made on the basis of disk accesses required for addition and deletion of attributes as well for queries.

The first technique, 'Directory Modification', has been developed to allow for arbitrary addition and/or deletion of attributes. The normalized disk accesses for the 'Directory Modification' method show a linear relationship with the fraction of records whose attributes are modified; the slope of this line is 2. Consequently, 'Directory Modification' is superior to generation of the BANG file from scratch for attribute modifications up to 50% of the records in the data file.

In practical examples where the number of attributes modified at a time is only a small fraction of data say less than 5%, 'Directory Modification' takes less than 10% of the disk accesses for regeneration of the BANG file, a significant reduction in the disk accesses.

During the development of 'Directory Modification' technique, some properties of the BANG file structure emerged; these have not been published in the literature but have the effect of drastically increasing the

complexity of developing and coding for the 'Directory Modification' method.

The data occupancy and disk accesses for range query for a data set depend on the ordering of the data. Permutation of the data set can change the regions defining the buckets. Directory splitting can result in buckets/regions that point to empty buckets, and this has to be checked. During deletion it is possible to end up with regions that cannot be merged with either a buddy, enclosing or enclosed regions and the region has to be merged with another region, that is a sibling or has a common boundary. This form of merging was not recognized previously.

One of the limitations of the 'Directory Modification' is that, for every operation we have to deal with the maximum number of attributes/dimensions, even though all the attributes may not be present for all records. This is also true in the case of 'Traditional' method. The 'Directory Modification' technique can only match the disk access efficiency of the 'Traditional' method which is inefficient for range/partial queries when a large number of records have zero values for the attributes.

The second technique 'Multiple BANG' files obviates the problems associated in dealing with databases where a large number of records have zero values for the attributes.

The generation of the 'Multiple BANG' file requires disk accesses comparable to 'Directory Modification' method. Consequently, this technique like the 'Directory Modification' is vastly superior to the 'Traditional' approach when the number of records whose attributes are modified is small.

'Multiple BANG' files show significant improvement over 'Traditional' BANG files in disk accesses for range and partial queries. The improvement is dependent on the fraction of the data searched for range/partial queries between the single file and the multiple BANG files accessed by the query. As the fraction of the data searched becomes smaller the improvements become more dramatic. The absolute reduction in disk accesses is enhanced as the range is increased, or the number of attributes in the partial query are decreased. Reduction in disk accesses greater than 50% are observed when the data search is 25% of the single BANG file.

The 'Multiple BANG' files approach is the superior method. It will be particularly useful for large attribute files, where the data is distributed over a combination of attributes. 'Directory Modification' approach will be useful if the data ends up with a full complement of attributes for all records after repeated modifications.

This thesis deals with the development and comparison of two new techniques for dynamic modification of attributes. Adequate testing of statistical variability

resulting from the data distribution needs to be studied in future work. This can be accomplished by averaging results over a large number of random realizations of attribute values for both the techniques. Non-uniform data distribution also needs to be tested.

In the case of 'Directory Modification' technique, range/partial queries should be performed to evaluate the effect of data occupancy on the disk accesses. This can then suggest an optimum percentage of attribute addition by 'Directory Modification' after which the BANG file needs to be regenerated from scratch.

This thesis presents the linked list structure in the case of 'Multiple BANG' files. Evaluation of data structures adapted to partial queries for a large number of files can enhance the applicability of this powerful approach.

BIBLIOGRAPHY

[BENT 75] Bentley, J.L. " Multidimensional binary search tree used for associative searching" Communications of the ACM 1975, 18(9), 509-517.

[BENT 79] Bentley,J.L., and Friedman, J.H. " Data structure and range searching" ACM computing surveys 1979, 11(4), 397-409.

[BENT 80] Bentley, J.L., and Maurer, H.A. "Efficient Worst-Case Data Structures for Range Searching" Acta Formatica 13, 1980, 155-168

[BURK 83] Burkhard,W.A. "Interpolation based index maintenance" Proc ACM symp Principles of Database Systems (1983), 76-89.

[CHAN 81] Chang Joe-Mei, Fu King-Sun. "Extended K-d Tree Database Organization: A Dynamic Multiattribute Clustering Method" IEEE Sept 1981, 284-290.

[CHUN 89] Chun,S.H., Hedrick G.E., Lu H., and Fisher D.D. "A Partitioning Method for Grid File Directories" IEEE Proc of the 13th Annual International Computer Software and Applications Conference 1989, 271-277.

[EDWA 63] Edward, H., and Sussenguth Jr. "Use of Tree Structures for Processing Files" Communications of ACM May 1963, 272-279.

[FAGI 79] Fagin,R., Nievergelt,J., Pippinger, N.,and Strong, H.R. "Extendible hashing - a fast access method for dynamic files" ACM Tran on Database Systems Vol4, No.3 (Sept 79), 315-344.

[FINK 74] Finkel,R.A., and Bentley, R.a. " Quad Trees - A data structure for retrieval on composite keys" Acta Informatica 4, (1974), 1-9.

[FRED 60] Fredkin, E. " Trie Memory" Communications of ACM 3, 9(Sept 60), 490-499.

[FREE 87] Freeston, M. " The BANG file : a new kind of grid file" Proc ACM SIGMOD (Dec 1987), 260-269.

[FREE 89a] Freeston, M. "Advances in the design of BANG

file" Third Internatoinal Conference on Foundations of Data Organisation and Algorithms, Paris, June 1989

[FREE 89b] Freeston, M. "A well behaved file structure for the storage of spatial objects" Symposium on the Design and Implementation of Large Spatial Databases Santa Barbara, California, July 1989.

[GOPA 80] Gopalkrishna, V., Madhavan, C. E. Veni. "Performance Evaluation of Attribute Based Tree Organization" ACM Transaction on Database Systems, Vol 5, No.1, March 1980, 69-87.

[GUTT 84] Guttman, A. "R Trees: A Dynamic Index Structure for Spatial Searching" ACM 1984 47-57.

[HINR 85] Hinrichs, K. " Implementation of grid file : Design concept and Exp" BIT 25, 3 (1985) 569-582.

[HUTF 88] Hutflesz, A., Six Hans-Werner., and Widmayer, p. "Twin Grid Files" ACM March 1988, 183-198.

[IYEN 88] Iyengar,S.S., Rao,N.S.V., Kashyp,R.L., and Vaishnavi, V.K. "Multidimensional data stuctures: Review and Outlook" Adv. in computers Vol 27(1988), 69 - 119.

[LEE 80] Lee,D.T., and Wong.C.K. "Quintary Trees : A file Structure for Multidimensional database systems" ACM-TODS vol 5, No.3, Sept 80, 339-353.

[LIAN 88] Lian,T.H. "Implementation and evaluation of BANG file structure" O.S.U. Master Thesis 1988.

[LITW 80] Litwin,w. "Linear Hashing: a new tool for file and table addressing" Proc. 6th Int. Conf. on Very Large Databases, 212-223.

[LUCK 78] Lucker, George. "A Data Structure for Orthogoanl Range Query" IEEE Sept 1988 28-33.

[NAGE 85] Nageswar Rao, S.V., Sitharama Iyengar, S.,and Veni Madhavan, C.E. "A Comparative Study of Multiple Attribute Tree and Inverted File Structures for Large Bilbliographic Files" Information Processing and Management Vol. 21, #5 1985, 433-442.

[NIEV 84] Nievergelt,H.J., and Sevcik,K.C. " The Grid File: An adaptable, symmetric multikey structure" ACM Trans on Database systems vol 9, No.1, (march 1984).

[OUKS 81] Ouksel, M., and Scheurmann,P. "Multidimensional B-Trees: Analysis of dynamic behavior" Bit 21 (1981) 401 - 418.

[SARI 87] Saritepe, H.N.A. "An Analytical of Grid File and k-d-B Tree Structures" O.S.U Master's Thesis 1987

[SCHE 82] Scheurmann,P., and Ouksel,M. " Multidimensional B- tree for associative searching in database systems" Inf. Systems 7, 2(1982), 123 - 137.

[TAMM 83] Tamminen, M. " On Search by address computation" Report HTKK-TKO-B56, Helsinki Univ of Tech Espoo 1983

APPENDIX

PROCEDURES TO ADD AND REMOVE ATTRIBUTES

This file has procedures related to the first technique, developed for ADDITION AND REMOVAL OF ATTRIBUTES. The technique is referred as 'Directory Modification' Main program is not included. Most of the variables are global, and these procedures are called from main program. Thus, the file can not be tested independently.

```
/*-------------------------------------------------------*/
PROCEDURE TRANS_ENTRY() IS USED TO MODIFY SINGLE ENTRIES
IN THE SUBDIRECTORY DURING THE ADDITION OF ATTRIBUTE.

PROCEDURE BACK_TRANS() IS USED  TO MODIFY SINGLE ENTRIES
IN THE SUBDIRECTORY DURING THE DELETION OF ATTRIBUTE.


MAJOR PROCEDURES :

INTERMEDIATE():  Converts the buddy regions into enclosing
                 regions, for a set of entries in a
                 subdirectory bucket.

MOD_SUBDIR()  :  Used during Addition of Attribute, to
                 modify subdirectory entries. Calls
                 TRANS_ENTRY() to modify single entry.

MOD_SUBDIR_DECR() :  Used during Deletion of Attribute to
                     modify subdirectory entries.  Calls
                     BACK_TRANS() to modify single entries.

/*-------------------------------------------------------*/



/* This procedure is used during the 'Addition of
   Attribute' to modify the subdirectory entries. Uses
   TRANS_ENTRY() for transformation of single entries. */

MOD_SUBDIR()
{
     int    numsub, i,j,k , level ;
     unsigned long reg, mask1,mask2 ;
     int   part_level ;    long   loc ,tloc ;
     BSUBDIR    temp ;
     NODE     *cell[MAXCELL] ;
```

```
        LEVEL[DIMENSION] = 0 ;
        mask2 = power(2,DIMENSION-1) ; /* mask2 = (2^DIM-1) */
        mask1 = power(2,DIMENSION) ;
        mask1 -= 1 ;          /*mask1 = (2^DIM) -1  */

        numsub = br_dir.header ;
        for(i = 0; i < numsub ; i++)
        {
            loc = (long)(i) * BLOCK_SIZE ;
            lseek(b_dr,loc,0) ;
            read(b_dr,&temp, sizeof(BSUBDIR)) ;
            j = temp.header ;
            for(k = 0; k < j ; k++)
            {
                reg = temp.cell[k].region ;
                level = temp.cell[k].level ;
                /* transform single entry */
                TRANS_ENTRY(&reg , &level , mask1, mask2) ;
                temp.cell[k].region = reg ;
                temp.cell[k].level = level ;
            }
           if(level > TOTAL_LEVEL)
                  TOTAL_LEVEL = level ;

          /* Convert the enclosing regions to buddy
                                regions, if possible */
           for(k = 0 ; k < j  ; k++)
                cell[k] = &temp.cell[k] ;
           do
           {
            for(k = 0; k < j; k++)
                FIND_UNIQUE_PAIR(cell[k],cell,j);
           } while((CHANGES1 == 1) || (CHANGES2 == 1) ) ;

           for(k = 0; k < j ; k++)
                COPYNODE(&temp.cell[k],cell[k]);

           SORTED(temp.header,temp.cell) ; /* sort the sub dir
                                                 entries */
           lseek(b_dr,loc,0) ;
           write(b_dr, &temp, BLOCK_SIZE) ;
          /* BCT_FETCH++ ;*/
        }
        part_level = 0 ;
        for(k = 0; k < DIMENSION ; k++)
          part_level = part_level + LEVEL[k] ;

        /*Modify the total LEVEL, and increase the DIMENSION*/
        LEVEL[DIMENSION] = TOTAL_LEVEL - part_level ;
        DIMENSION ++ ;

    }   /* END OF ROUTINE -- MOD_SUBDIR() */
```

```
/-------------------------------------------------------/


/* Called from MOD_SUBDIR() to transform the single
   entries in the subdirectory bucket. For a given pair of
   region and level, returns the transformed region & level
   .
*/

TRANS_ENTRY(reg, level, mask1, mask2)
unsigned long   *reg , mask1, mask2;
int  *level ;
{
     unsigned long    b, c,newreg ,incr = 1;
     int     i, j  ;

     newreg = (*reg) & mask1 ; /* save th unaffected lowest
                                  order bits*/

     for(i = 0; i < (*level)/DIMENSION ; i++)
     {
        mask2 <<= 1 ;  /*set mask2 at the start of cycle */

          for(j = 0; j < DIMENSION ; j++)
          {
              mask2 <<= j ;
              b  = (*reg) & mask2 ; /* choose the intended
                                       bit */
              b <<= (i + incr) ;
              newreg = newreg | b ;/* set the chosen bit in
                                      the right position
            }                         of modified region */
      }

      *reg = newreg ;              /* transformed region
      if(((*level) % DIMENSION) == 0)        and level */
         *level += (i - 1) ;
      else
         *level = *level + i ;

}    /* END OF ROUTINE  -- TRANS_ENTRY() */


/*-------------------------------------------------------*/


/* Intermediate step to convert the buddy regions into
   enclosing regions. Calls the routine MAKE_ENCLOSE(), for
   the conversion. All other entries are left unchanged.*/

INTERMEDIATE(type)
int   type ;   /* increase/decrease dimension */
{
```

```
int          i,j, totroot,nsub_entry,exist ,level,ptr;
int          pos ,k ,changes,push ;
BSUBDIR      temp ;
unsigned long   reg ;
long          loc  ;

totroot = br_dir.header ;
for(i = totroot -1 ; i >=0 ; i--)
{
      nsub_entry = br_dir.cell[i].header ;
      SET_HEAD(i,&loc) ;
      read(b_dr, &temp, sizeof(BSUBDIR)) ;
      BCT_FETCH++ ;
      j = nsub_entry - 1 ;
      while(j >= 0)
      {
         changes = 0 ;
         level = temp.cell[j].level ;
         reg = temp.cell[j].region ;


         if(type == 1)   /* increase the dimension */
         {
            if(level > DIMENSION)
            MAKE_ENCLOSE(&temp,reg,level,&changes,j) ;
         }
         else
         if(type == -1) /* decrease the dimension */
         {
            if(level >= DIMENSION)
            MAKE_ENCLOSE(&temp,reg,level,&changes,j) ;
         }

         if(changes == 1)/* buddy or the components of
                            buddy are found, i.e level
                            is modified, so sort, but
         {                  but do not change index*/
            PART_SORT(j,&temp,&push) ;
            if(push == 0)
               j = j - 1 ;   /* position was not
                                changed, goto next*/
         }
         else             /* buddy not found, go to
                             next entry */
            j = j -1 ;

    }
     SORTED(temp.header,temp.cell) ;
     DELETE_EMPTY(&temp) ; /* delete empty regions */
     lseek(b_dr,loc,0);
     write(b_dr, &temp, BLOCK_SIZE) ;
     BCT_FETCH++ ;
     }
}     /* END OF ROUTINE  -- INTERMEDIATE()   */
```

```
/*-------------------------------------------------------*/


/* For a given entry,  this procedure finds the exact
   buddy, or the components of the buddy. If found,
   decreases the level, in the entry to make it enclosing
   region */

    MAKE_ENCLOSE(temp,reg,level,changes,j)
    BSUBDIR   *temp ;
    unsigned long    reg;
    int     level, j, *changes ;
    {
           int      higher, blevel ,found = 0 ;
           unsigned long   breg ;

           higher = power(2,level-1); /* higher buddy */
           if(reg < higher)           /* lower buddy */
           {
                 buddy(reg,level,&breg) ;
                 blevel = level ;
                 found =
                 sch_sub_same_level(breg,blevel,*temp,j);
                 if( found > -1)   /* found exact buddy */
                 {
                     temp->cell[j].level = level - 1 ;
                     *changes = 1 ;
                 }
                 else       /* find components of buddy */
                 {
                     MAINFLAG = 1 ;
                     MAINFLAG = CHK-SUB-RECUR(breg,blevel,
                                        level,*temp ,j);
                     if(MAINFLAG == 1)
                     {
                         temp->cell[j].level = level -1 ;
                         *changes = 1 ;
                     }
                  }
              }
       }  /* END OF ROUTINE -- MAKE_ENCLOSE() */


/*-------------------------------------------------------*/


/* Finds the buddy region (breg) for a given region (r)*/

buddy(r,l,breg)
unsigned long     r, *breg ;
int               l  ;
{
       unsigned long    max  ;
```

```
        max = power(2, l-1) ;
        if(r >= max)
           *breg = r - max ;
        else
           *breg = r + max ;
}    /* END OF ROUTINE -- buddy()  */

/*-------------------------------------------------------*/

/* Checks the subdirectory recursively to see if the
   components of the buddy are present. If all components
   are present, returns 1 else -1.    */

CHK_SUB_RECUR(breg,blevel,level,temp,j)
unsigned long    breg ;
int              blevel ,level, j ;
BSUBDIR          temp ;
{
        unsigned long   i ;

        if(blevel >= (level + DIMENSION))
            MAINFLAG = -1 ;
        else
        if( !sch_sub_high_level(breg,blevel,temp,j) )
        {
            CHK_SUB_RECUR(breg,blevel+1,level,temp,j) ;
            if(MAINFLAG != -1)
            {
               i = power(2,blevel) ;
               CHK_SUB_RECUR(breg + i, blevel+1,level,
                                              temp,j) ;
            }
        }
        return(MAINFLAG) ;
 }   /* END OF ROUTINE  -- CHK_SUB_RECUR()  */

/*-------------------------------------------------------*/

/* This routine searches for a buddy entry(breg,blevel)
   within the same level in a subdirectory bucket . If
   found returns the position of the buddy  in the
   subdirectory else returns -1. */

sch_sub_same_level(breg,blevel,temp,j)
int              blevel ,j;
unsigned long    breg ;
BSUBDIR          temp ;
{
      int        i ,h ;
```

```
        h = temp.header ;
        for( i = j+1 ; i < h ; i++)
        {
            if((temp.cell[i].level == blevel) &&
                        (temp.cell[i].region == breg))
                return(i) ;
         }
         return(-1) ;
}     /*  END OF ROUTINE -- sch_sub_same_level()  */

/*-------------------------------------------------------*/


/* Search for an entry in the subdirectory,  at higher
   levels, if found return 1 else return 0 */

sch_sub_high_level(r,l,temp,j)
int    l,j ;     /* j is the position of the entry in the
                    subdir temp */
unsigned long     r ;
BSUBDIR           temp ;
{

   int        i ,total ;

   total = temp.header  ;
   for( i = j+1 ; i < total ; i++)
   {
       if( (temp.cell[i].region == r) &&
                            (temp.cell[i].level == l))
           return(1) ;
   }
   return(0) ;
}    /* END OF ROUTINE  -- sch_sub_high_level() */



/*-------------------------------------------------------*/


/* Searches for a region in the given subdirectory 'temp'.
   If found, returns the position of the region, else -1 */

find_pos_in_sub(breg,j, temp)
unsigned long    breg ;
int              j ;
BSUBDIR          temp ;
{
     int  i, total ;

     total = temp.header ;
     for( i = j ; i < total ; i++)
     {
         if(temp.cell[i].region == breg)
```

```
                return(i) ;
        }
        return(-1) ;

}       /* END OF ROUTINE -- find_pos_in_sub()  */

/*-------------------------------------------------------*/

/* Sets the read head to the position from where a
   subdirectory needs to be read, pointed by 'i'th  root */

SET_HEAD(i,loc)
int     i ;
long    *loc ;
{
        int    ptr;

        ptr = br_dir.cell[i].ptr ;
        *loc = (long)ptr * BLOCK_SIZE ;
        lseek(b_dr, *loc , 0) ;
}    /* END OF ROUTINE -- SET_HEAD() */

/*-------------------------------------------------------*/

/* Within a subdirectory checks recursively if the entries
   that make up buddy exist, if they exist,level is
   decreased depending upon the region number. changes is
   set to 1,if level is modified   */

RECUR(breg,blevel,level,j,less,temp,changes)
unsigned long    breg ;
int              level,blevel,j,less, *changes ;
BSUBDIR          temp ;
{
      int   pos ;

       *changes = 0 ;
       MAINFLAG = 1 ;
       MAINFLAG = CHK_SUB_RECUR(breg,blevel,level,temp,j) ;
       if(MAINFLAG == 1)
       {
           pos = find_pos_in_sub(breg,j,temp) ;
           if(less == 1)
           {
               temp.cell[pos].level  -= 2 ;
               *changes = 1 ;
           }
       }
}    /*  END OF ROUTINE -- RECUR()  */
```

```
/*-------------------------------------------------------*/

/* Sorting is done within the subdirectory only on the
   affected Entries. Rest are left alone */


PART_SORT(j, temp,push)
int    j ;      /* position at below are left unchaned */
int    *push ;  /* tells whether the entry was relocated or
                   not */
BSUBDIR     *temp ;
{
      int    jlevel ,i ,jheader, jptr;
      unsigned long   jregion   ;


       *push = 0 ;
      jlevel = temp->cell[j].level ;
      jregion = temp->cell[j].region ;
      jheader = temp->cell[j].header ;
      jptr    = temp->cell[j].ptr    ;
      for(i = 0; i < j ; i++)
      {
          if(temp->cell[i].level == jlevel)
          {
              if(temp->cell[i].region == jregion)
              {
                  MESSAGE(5) ;
                  exit(0) ;
              }
              else
              if(temp->cell[i].region > jregion)
              {
                 PUSH_DOWN(temp,i, j, jregion,
                                      jlevel,jheader,jptr);
                 *push = 1 ;
                 break ;
              }
           }
           else
           if(temp->cell[i].level > jlevel)
           {
                PUSH_DOWN(temp,i, j,
                          jregion,jlevel,jheader,jptr);
                *push = 1 ;
                break ;
           }
       }
  }  /* END OF ROUTINE  -- PART_SORT()   */
```

```
/*-------------------------------------------------------*/

/* In the subdirectory 'temp', entries below the position
   'i' up to position 'j' are pushed down by one position.
   The ith entry is replaced by given jregion, & jlevel */


 PUSH_DOWN(temp,i,j, jregion,jlevel,jheader,jptr)
 int     i, j ,jlevel,jheader,jptr;
 unsigned long    jregion ;
 BSUBDIR    *temp;

 {
     int     pos ;

     for(pos = j - 1 ; pos >= i; pos-- )
     {
         temp->cell[pos+1].region = temp->cell[pos].region;
         temp->cell[pos+1].level  = temp->cell[pos].level;
         temp->cell[pos+1].header = temp->cell[pos].header;
         temp->cell[pos+1].ptr    = temp->cell[pos].ptr;
     }

     temp->cell[i].region = jregion ;
     temp->cell[i].level = jlevel  ;
     temp->cell[i].header = jheader ;
     temp->cell[i].ptr = jptr ;

}    /* END OF ROUTINE  -- PUSH_DOWN()   */


/*-------------------------------------------------------*/


/* This procedure is used during DELETION OF ATTRIBUTE to
   decrease the dimension of the existing problem. Calls
   BACK_TRANS() for transforming single entries. All the
   entries are assumed to be in one subdirectory */

MOD_SUBDIR_DECR()
{
    int     numsub, i, j, k, nattrib ,level;
    BSUBDIR    temp;
    long       loc ;
    unsigned long   reg, mask ;
    NODE   *cell[MAXCELL] ;

    mask = power(2, DIMENSION-1) ;
    mask -= 1 ;    /* mask = (2**(DIM -1)) - 1  */
    nattrib = 0;
    nattrib = 3 ;
```

```
    /*DELETE_EMPTY() ;*/
    numsub = br_dir.header ;
    for(i = 0; i < numsub ; i++)
    {
         loc = (long)(i) * BLOCK_SIZE ;
         lseek(b_dr, loc, 0) ;
         read(b_dr, &temp, sizeof(BSUBDIR)) ;
         BCT_FETCH++ ;
         for(k = 0; k < temp.header ; k++)
         {
             reg = temp.cell[k].region ;
             level = temp.cell[k].level ;
             BACK_TRANS(&reg, &level, mask) ;
             temp.cell[k].region = reg ;
             temp.cell[k].level = level ;
         }
         if(level < TOTAL_LEVEL)
             TOTAL_LEVEL = level ;
         DIMENSION -= 1 ;   /* decrease the dimension */


        /* convert enclosing regions in the subdirectory
           into buddy regions if possible */
        j = temp.header ;
        for(k = 0 ; k < j  ; k++)
             cell[k] = &temp.cell[k] ;
        do
        {
          for(k = 0; k < j; k++)
             FIND_UNIQUE_PAIR(cell[k],cell,j );
        }while(( CHANGES1 == 1) || (CHANGES2 == 1)) ;

        for(k = 0; k < j ; k++)
             COPYNODE(&temp.cell[k],cell[k]);

         SORTED(temp.header, temp.cell) ;
         lseek(b_dr, loc, 0) ;
         write(b_dr, &temp, BLOCK_SIZE) ;
         BCT_FETCH++ ;
         printf("\n# OF PAGES ACCESSED TO DECREASE THE DIM
         = %d\n",BCT_FETCH);
     }
}    /* END OF ROUTINE -- MOD_SUBDIR_DECR() */


/*-------------------------------------------------------*/


/*  This routine is used to transform the single entries
    during the Deletion of Attributes, called from
    MOD_SUBDIR_DECR(). Right now it is only set to decrease
    the dimension by one.  Needs slight modifications for
    arbitrary number of attributes. */
```

```
BACK_TRANS(reg,level, mask)
int    *level ;
unsigned long     *reg, mask ;
{
    unsigned long   new_reg, b, c ,decr, treg;
    int      i ;

    treg = *reg ;
    decr = 1 ;          /* decrement dimension by 1 */
    new_reg = (treg) & mask ;

    for(i = 0; i < (*level)/DIMENSION; i++)
    {
        b = treg >> (i+decr) ;
        mask  <<= DIMENSION - 1 ;
        c = b & mask ;
        new_reg = new_reg | c ;
   }
   *level  -= i  ; /* decrement the level by the #of times
                      it has gone through the loop */
   *reg = new_reg ;  /* modify the region     */
}    /*    END OF ROUTINE -- BACK_TRANS()   */


/*-------------------------------------------------------*/


/* After removing about 35 to 40% of records, there are
   some empty buckets.  This routine, tries to find an
   enclosing region and merges with it. Entry corresponding
   to empty region is removed. */

DELETE_EMPTY(temp)
BSUBDIR    *temp ;
{
     NODE     enode ,big;
     int      i ,p , pos ,header, found, total;

     total = temp->header ;
     for( i = total - 1 ; i >= 0 ; i--)
     {
          header = temp->header ;
          if((temp->cell[i].header) == 0)
          {
             enode.region = temp->cell[i].region ;
             enode.level  = temp->cell[i].level  ;
             enode.header = temp->cell[i].header ;
             enode.ptr = temp->cell[i].ptr ;
             for(p = i-1 ; p>=0; p--)
             {
                found = 0 ;
```

```
                if((temp->cell[p].level) < enode.level)
                {
                   big.region = temp->cell[p].region ;
                   big.level  = temp->cell[p].level  ;
                   big.header = temp->cell[p].header ;
                   big.ptr = temp->cell[p].ptr ;
                   found = IS_INCLUDE(enode,big ) ;
                   if(found)
                   {
                     RETURN_DT_TO_AVAIL(temp->cell[i].ptr);
                       for(pos = i+1 ; pos < header; pos++)
                       {
                          temp->cell[pos-1].region =
                                  temp->cell[pos].region ;
                          temp->cell[pos-1].level  =
                                  temp->cell[pos].level  ;
                          temp->cell[pos-1].header =
                                  temp->cell[pos].header ;
                          temp->cell[pos-1].ptr =
                                  temp->cell[pos].ptr ;
                       }
                       temp->header-- ;
                       br_dir.cell[0].header-- ;  /*modify
                                                   root dir */
                       break ;
                    }
                    else
                     fprintf(fr,"\n NO merging Candidate");

                }    /* end of if*/
             }
          }
    }       /* end of for */
}       /*  END OF ROUTINE -- DELETE_EMPTY()   */
```

VITA

Nalini T. Hosur

Candidate for the Degree of

Master of Science

Thesis: DYNAMIC ADDITION AND REMOVAL OF ATTRIBUTES IN BANG FILES

Major Field: Computer Science

Biographical:

Personal Data: Born in Karnatak, India, August 1954, daughter of Late Smt. and Shri. R B. Sonnad, daughter-in-law of Smt S L. Hosur and Late Shri L T. Hosur.

Education: Received Bachelor of Science Degree in Physics from University of Bombay in March, 1974; Master of Arts in Industrial Psychology from Karnatak University in Feb, 1981; Doctor of Philosophy in Social Psychology from Karnatak University in Feb, 1981; completed the requirements for the Master of Science degree at Oklahoma State University in May, 1991.

Professional Experience: Lecturer, Department of Psychology, Karnatak University, Karnatak, India, March, 1981, to May, 1985.