COMPRESSION TECHNIQUES FOR THE GRID FILE

BY

SEI HOON CHUN

Bachelor of Science in Electrical Engineering

Hanyang University

Seoul, Korea

1985

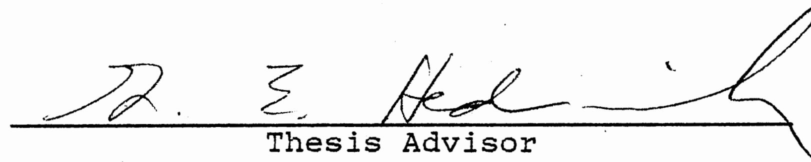
Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1988

Thesis
1988
C559c
cop.2

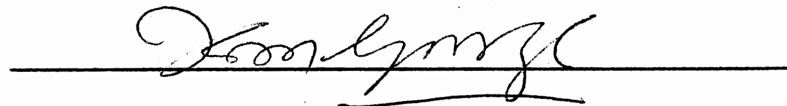
Oklahoma State Univ. Library

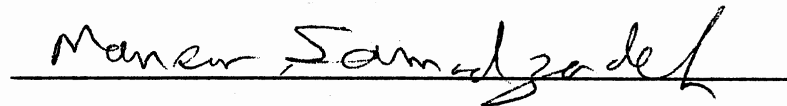
COMPRESSION TECHNIQUES FOR THE GRID FILE

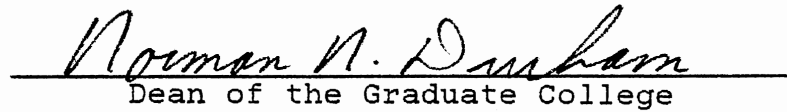
Thesis Approved:



Thesis Advisor







Dean of the Graduate College

1318988

ACKNOWLEDGMENTS

I want to show my sincere respect and the appreciation to my major advisor Dr. George E. Hedrick, for his guidance on this study and also for his warm encouragement during my academic career in Oklahoma State University. I would like to extend my thanks to Dr. K. M. George and Dr. M. Samadzadeh for serving as members of my graduate committee.

I would like to express my appreciation for helpful discussions of Dr. David L. Nofziger. Also, I am grateful to Charles Ellis, Michael Gibson, and Judas Riley for proofreading the thesis.

Most of all, I would like to express my thanks to my parents, Yung Il Chun, and Jung Eui Chun for their love and support during my thesis writing.

TABLE OF CONTENTS

Chapter | Page

I. INTRODUCTION 1

II. REVIEW OF HASHING TECHNIQUES 4

Introduction 4
Single-key hashing method 5
Multi-key hashing method 13

III. DATA COMPRESSION METHOD 20

IV. THE GRID FILE 30

Introduction 30
Adaptability of the grid file 36
Splitting policy 38
Merging policy 43

V. COMPRESSION TECHNIQUES APPLIED TO THE GRID FILE STRUCTURE 50

Directory compression 50
Bucket compression 61

VI. SUMMARY, CONCLUSION, AND SUGGESTED FUTURE WORK . . 67

BIBLIOGRAPHY 70

APPENDICES 74

APPENDIX A - BUCKET OCCUPANCY OF THE GRID FILE STRUCTURE 74

APPENDIX B - TOTAL FILE COMPRESSION USING THE MIDPOINT OF RECORD PARTITION METHOD 76

LIST OF FIGURES

Figure Page

2-1. Initial Structure of a Dynamic Hashing File. . . 5

2-2. The Dynamic Hashing Structure after the Third Partition Operation. 7

2-3. Extendible Hashing with Total Depth d=2. 8

2-4. Extindible Hashing with Total Depth d=3. 9

2-5. Collision of Data Bucket 0 11

2-6. Extended Bucket Storage by h_1 Function 12

2-7. H = (h_o, h_o) in a Two-Dimensional MLDH File. . . 16

2-8. H = (h_1, h_o) in a Two-Dimensional MLDH File. . . 16

2-9. H = (h_1, h_1) in a Two-Dimensional MLDH File. . . 17

3-1. Suppresion of Repeated Character Method. 22

3-2. Huffman Coding Tree. 23

3-3. Encoding and Decoding of Huffman Coding. 24

3-4. 2/4 Code 26

4-1. A Search for a Record in the Grid File 34

4-2. Double Levels of the Directory 35

4-3. Grid Directory and Bucket. 36

4-4. Splitting of a Directory 37

4-5. Subsequent Splitting of a Directory. 38

4-6. Buddy System and Neighbor System 40

4-7. Finding Levels of the Scale. 41

4-8. Splitting of a Directory Page. 43

Figure | Page

4-9. Deadlock in a Neighbor System 44

4-10. Partition of a 3-Dimensional Directory. 45

4-11. A Deadlock State in a 3-Dimensional Directory . 46

4-12. Deadlock Prevention Algorithm 47

4-13. Merging Algorithm 48

5-1. 4 partitions Are Needed in a 2-Dimensional Grid File Structure where Bucket Capacity b = 1. . . 51

5-2. Partition at the Midpoint of Record in 2-Dimensional Grid File Structure 52

5-3. Cranston's Partitioning and Merging Method. . . 53

5-4. Single-Level Biased Data Grid File Comparison . 55

5-5. Double-Level Biased Data Grid File Comparison . 56

5-6. Double-Level Random Data Grid File Comparison . 57

5-7. Grid Partition Methods. 59

5-8. Compression of Bucket Record. 62

5-9. Total File Size Variation using a Biased Data File 65

6-1. Total File Size Variation using a Randomly Generated Data File 68

A-1. Grid File Bucket Occupancy using a Randomly Generated Data File 74

A-2. Grid File Bucket Occupancy using a Biased Data File. 75

B-1. Total File Size Variation using a Randomly Generated Data File 76

B-2. Total File Size Variation using a Biased Data File. 77

CHAPTER I

INTRODUCTION

A variety of file structures have organizational methods to aid data management. For example, programs using sequential files, tree structured files, or hash files manipulate their data with a single primary key. Traditional, static hashing methods frequently outgrow overflow buckets by continuous insertions. This problem has been somewhat mollified by implementing dynamic hashing methods [FAGI79, RAMA85, LARS87, TAMM82].

Dynamic hashing methods do not suffer severe degradation from overflow bucket storage. They achieve this by reorganizing their shape continuously with the insertion of data. These structures adapt their structures dynamically without any loss of efficiency. File structures that provide multikey access to data are of great interest in various situations. For example, the organization of a physical database for efficient query needs two or three dimensions without favoring a certain dimension. To achieve this goal, several multikey file structures that avoid the deficiencies of inverted files have been suggested. These strategies generally combine all fields into a single access path so that the structure treats every field as a primary

key. None of these single structures can be optimal under all circumstances.

The optimal structure is the bitmap representation [VALL76]. The bitmap representation of a record reserves one bit for each possible record in the total data space, and at the same time represents whether the record is present in the file or not. The bitmap representation appears to be an ideal solution for the multikey file structure because it does not require reorganization of the structure. However, the bitmap approach does present a major inconvenience since it needs an impracticably large amount of storage space.

One attempt to deal with the bitmap storage problem was proposed by Nievergelt [NIEV84]. By introducing a dynamic directory, Nievergelt suggests a grid file structure, which is both symmetric and adaptive for multikey access. Symmetry means that every field is a primary key. Adaptability means that the directory adapts its structure both dynamically and automatically to the bucket. Consequently, the directory size responds even at such primitive level operations as inserting, deleting, and finding in an automatic and dynamic manner. A further advantage of this structure is that the expected access time for a retrieval should be the same over the entire file by the adapted directory.

Using the Nievergelt structure, analyses [REGN85, SARI87], applications [HINR83, HINR85], and also

concurrencies [HAN88, SALZ86] have been researched actively. By utilizing two directory levels, the grid file structure is not only good for a highly dynamic multikey structure, but, it can conquer the storage problem since the grid structure handles a large amount of data.[1]

By using a simplified recombination method proposed by Cranston [CRAN75], a new directory partition method is presented. The new partition method improves the worst-case numbers of disk access. Also bucket size for an actual record is reduced using various compression techniques.

This thesis presents the ramifications of this research and its practical applications. Chapter II previews both several primary key dynamic hashing methods and multi-key hashing methods. Chapter III previews various data compression methods for implementing the grid file structure. The general grid file structure and primitive operations are explained in Chapter IV. Chapter V outlines the directory compression method and the organization of the data compression method applied to the grid file structure. Chapter VI concludes the overall structural analysis and evaluates the expected cost and complexity of the storage.

---

[1] The comparison of one level and two levels of grid diectory is done by Han [HAN88].

CHAPTER II

OVERVIEW OF HASHING TECHNIQUES

Introduction

Hashing is one of the fastest techniques for managing direct access files. The address calculation technique is so simple that retrieval, insertion, and deletion of records can be done using the hash function. Compared to tree structured techniques which require several accesses, the hashing technique allows the retrieval of a record in one access, when a file or a table is almost static. If the file, or the table, is dynamic, then since the performance of hashing is possibly prone to disintegration by a collision of data, the performance of tree structures is much better.[1] Several dynamic hashing methods have been suggested for handling a dynamic set of data [FAGI79, LARS87, LITW80]. These hashing algorithms work efficiently for the dynamic files. However, all of the above structures are ranges of hash functions are linear. When the records have suitable only for one-dimensional data sets in which

[1] Use of the hashing technique required both the size of the file and the physical storage space allocated for the file to be known in advance in order to avoid collision.

the several attributes, it is necessary to have a multi-dimensional data structure for handling all attributes of a record efficiently. One main advantage of a multi-dimensional data structure is that it can handle partial match queries very easily. Several multi-dimensional dynamic hashing methods [BURK83, HUAN85, OUKS83, TAMM82] have been developed from one dimensional hashing methods.

Single-key Hashing Methods

Dynamic hashing [LARS87] is based on conventional hashing, except that the allocation of storage space can be increased and decreased dynamically without reorganizing the file. The index structure of the dynamic hashing is reorganized dynamically, according to the number of records actually stored in the file. The initial index and bucket

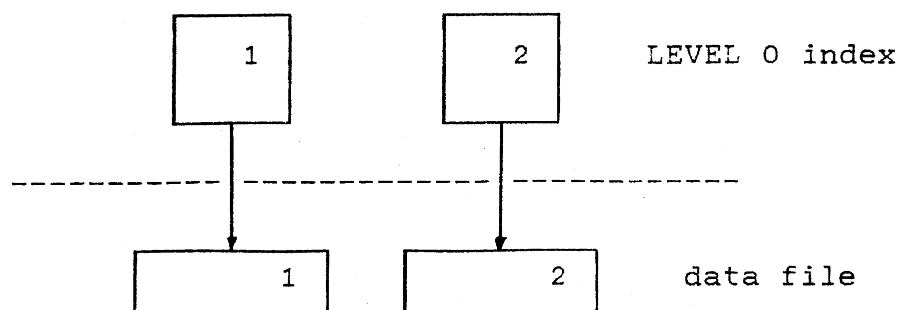


Figure 2-1. Initial Structure of a Dynamic Hashing File.

storage structure of the dynamic hashing is given in Figure 2-1. External nodes represented by the box shapes contain actual data bucket addresses. If the record to be inserted is represented as R, then the initial hash function $H_o(R)$ is used for addressing. $H_o(R)$ is a standard hashing function, so the value of $H_o(R)$ separates the level 0 indexes and puts the record R into the appropriate bucket storage. For example, the hashing function is

$$H_o(R) = (R \bmod 2) + 1$$

in Figure 2-2. As a result, if a record R has a even value, then that record stores in the bucket storage under index 1; otherwise, it stores in the bucket storage under index 2. By inserting data continuously, the bucket storage overflows. Then, partitioning of the data bucket becomes necessary, in order to contain this overflow. Storage space for the new bucket is allocated and the records are divided between two buckets. At the same time, the index of the separated bucket must be updated. Figure 2-2 shows the dynamic hashing structure after three partitionings.

Retrieval of a record, R, can be accomplished using $H_o(R)$ and $B(R)$; where, the hash function, $B(R)$, is a uniquely defined pseudo-random number associated with R. This function is performed by the equation

$$B(R) = (b_o, b_1, b_2, \ldots, b_i, \ldots) \quad b_i \in \{0, 1\}.$$

The root of a search tree can be located using $H_o(R)$. The searching for the record R in the index tree structure can be determinated using $B(R)$. The determination of left or

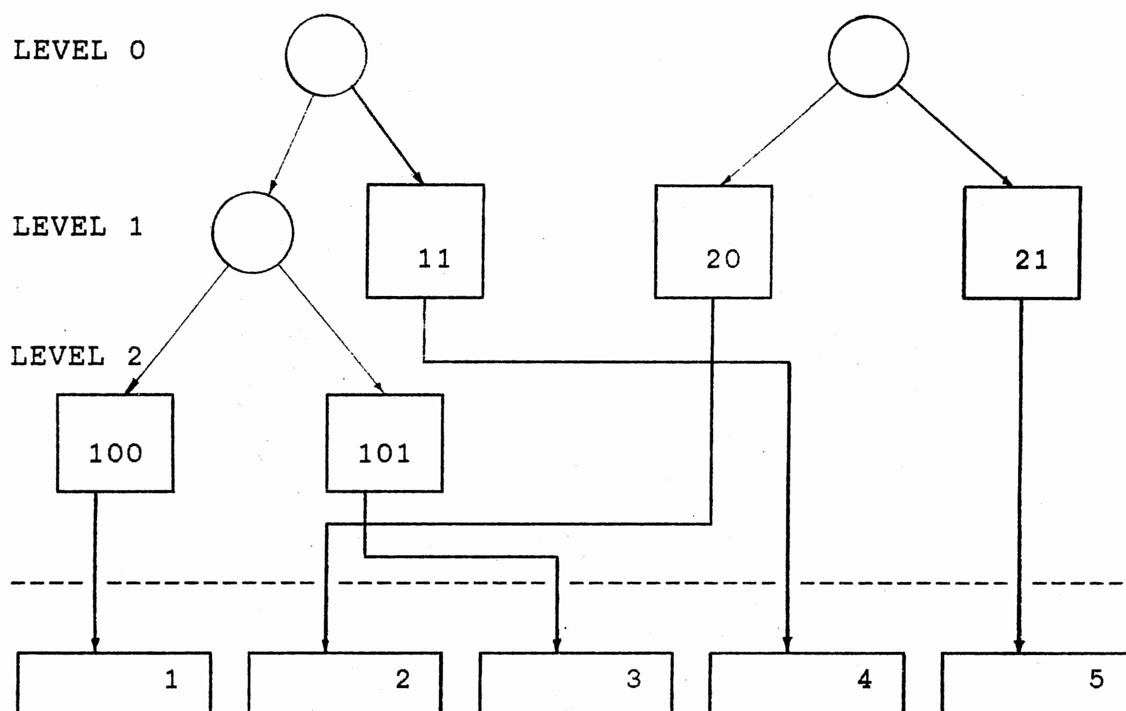


Figure 2-2. The Dynamic Hashing Structure after the Third Partition Operation.

right subtree paths is done using b_i; where, i = level + 1. The pseudo-random function, B(R), balances the tree yielding short search paths.

If the number of records in two brother buckets becomes less than or equal to the capacity of one bucket, these two brother buckets merge into one bucket. Two buckets are brothers if the two external nodes which point to these buckets have the same father node. At the same time, that these two brother external nodes are being deleted, the internal node associated with these two external nodes is

itself changed to an external node. The new external node points to the new bucket by the merger of the brother buckets. The dynamic hashing structure maintains its structure according to insertions and deletions without the use of overflow chains. When index trees are maintained in main storage, a maximum of one access to the storage structure is guaranteed. However, this performance depends upon the index structure being small enough to fit into main storage.

Extendible hashing [FAGI79] is developed from the randomly grown radix search tree [FRED60]. It only has a directory structure and a bucket structure. The structure contains a header which represents the total depth, d, of

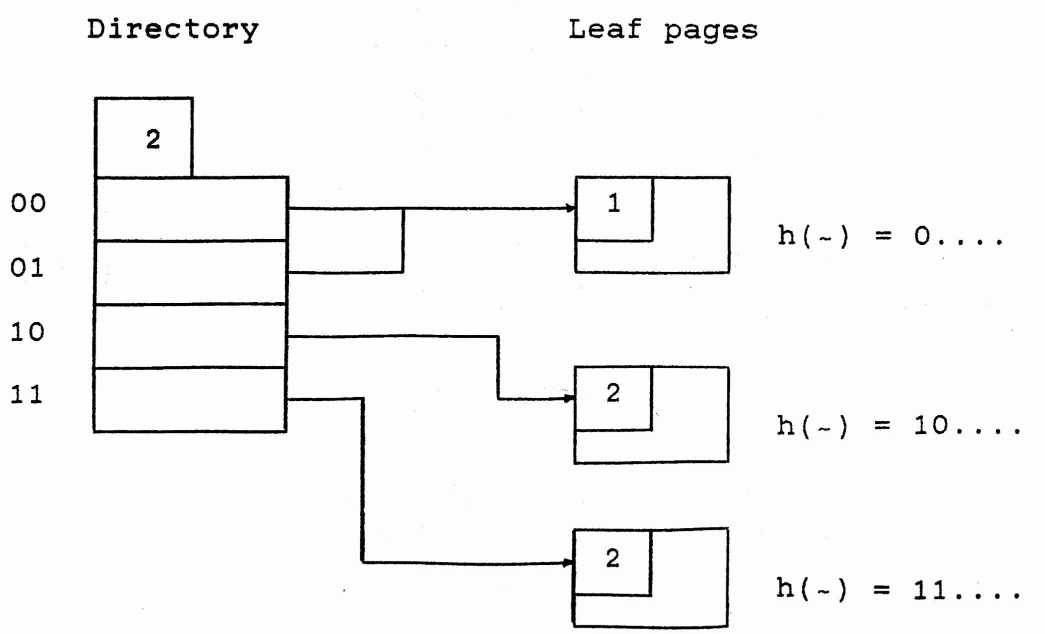


Figure 2-3. Extendible Hashing with Total Depth d = 2.

the directory. Figure 2-3 shows the case, where d = 2, at the structure. The directories with pointers to the bucket storage follow the header. The number of directories is 2^d because directories are doubled each time directory partitioning is invoked. Each leaf page has a header that contains a local depth d'; consequently, all keys in the local page have the same first d' bits. The total depth, d, of the directory is the maximum of the local depth, d', of all the leaf pages. When the leaf page overflows, the first reaction is to check the local depth, d', with depth d. If

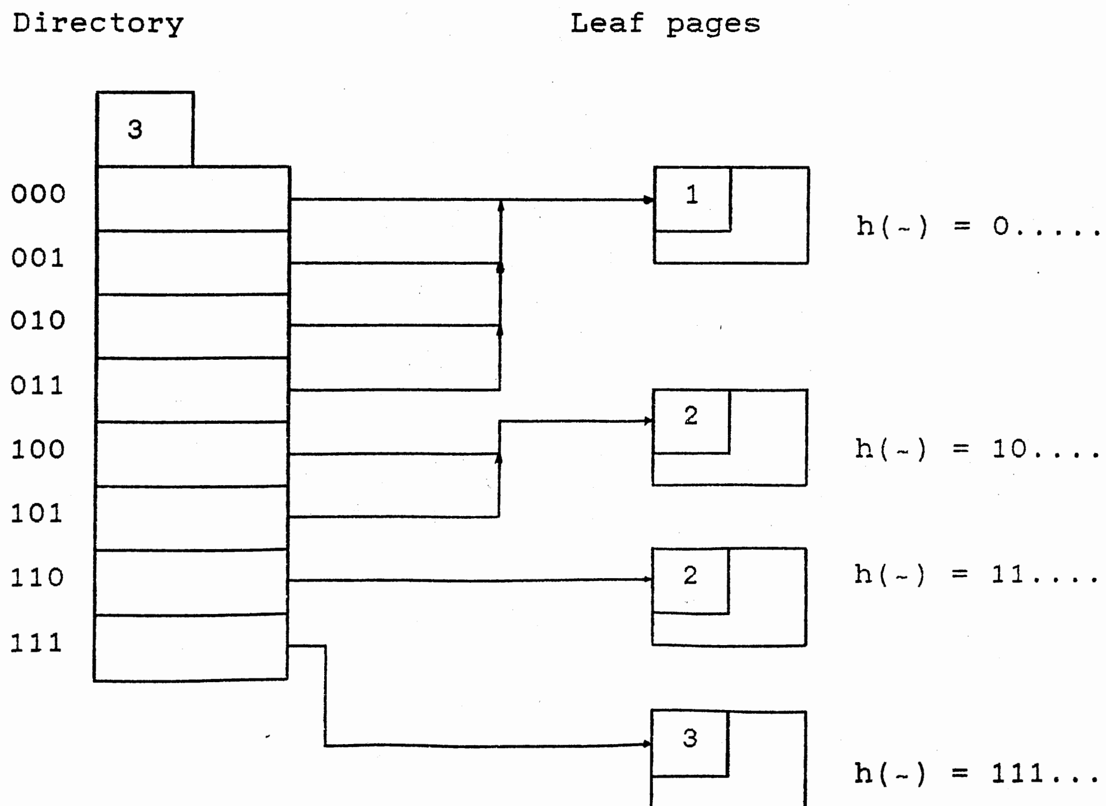


Figure 2-4. Extendible Hashing with Total Depth d = 3.

d' < d, then only one new bucket is created; there is no directory doubling. The structure shown in Figure 2-4 is the double state of the directory shown in Figure 2-3.

To retrieve the record, R, from the extendible hashing structure, h(R) must be calculated and its first d bit must be found. Using the address computation method, the location of the directory which has the same first d bit patterns can be ascertained. Then, the bucket can be found using the pointer from the directory. A maximum of two-disk accesses is guaranteed.

The cost of directory doubling is not as expensive as it appears because the actual data buckets are not changed. Also, the doubling of a directory is done using the buddy system [PECH82], so the merging of buckets occurs without external access.

Linear hashing [LITW80] suggested by Litwin is another dynamic hashing techniques. The hashing function

$$H = (h_0, h_1, h_2, h_3, \ldots, h_i, \ldots),$$
$$\text{where } 0 \leq i < \infty$$

grows as record insertion makes the data in a bucket collide. To illustrate this principle, u is the total key space and N is the total number of buckets. Hence, in the equation

$$h_0 : u \rightarrow \{0, 1, \ldots, N - 1\}$$

the initial hashing function for loading is denoted by h_0. The hashing functions $h_1, h_2, h_3, \ldots, h_i, \ldots$ are called split functions of initial function h_0 if they satisfy the

following rules:

$$h_i : u \rightarrow \{ 0, 1, \ldots, 2^i N-1 \}$$

For any u either:

$$h_i(u) = h_{i-1}(u)$$

or

$$h_i(u) = h_{i-1}(u) + 2^{i-1}N$$

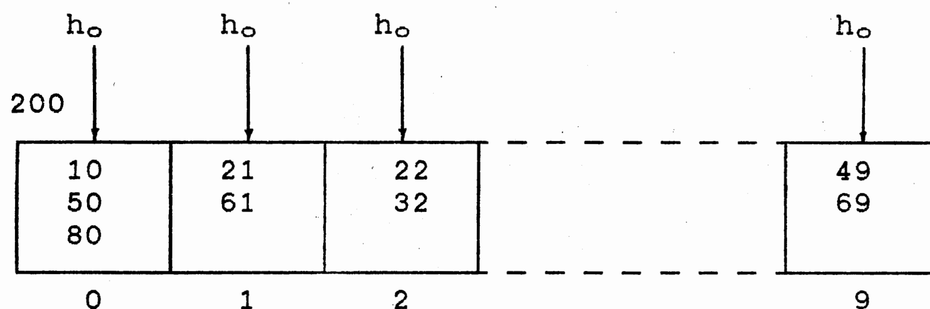


Figure 2-5. Collision of Data Bucket 0.

For example, the initial state of a data bucket is made by the function h_0 where the number of data items, N, is 10 (N=10) and bucket capacity, b = 3, is shown in Figure 2-5. Also, for this diagram, there is no overflow bucket to allow highly dynamic structures. If the next datum inserted is 200, then bucket 0 overflows. In this case, the hash

function, h_o, is changed:

$$\text{if } h_o(u) = 0, \text{ then}$$
$$h(u) = h_1(u)$$
$$\text{else } h(u) = h_o(u);$$

where, $h_o(u) = u \bmod N$, and $h_1(u) = u \bmod 2N$. The bucket content after splitting is shown in Figure 2-6.
As a result, the linear hashing scheme increases and decreases its bucket storage dynamically according to the incoming data.

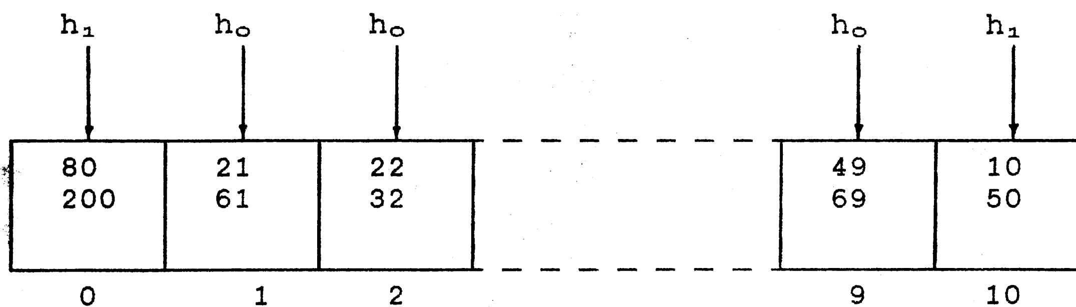


Figure 2-6. Extended Bucket Storage by h_1 Function.

To retrieve a record R, the following algorithm is used for calculating the bucket index.

$$idx = h_{LEVEL}(R)$$
$$\text{if } idx < NEXT, \text{ then}$$
$$idx = h_{LEVEL+1}(R);$$

The two variables LEVEL and NEXT are defined as:

- NEXT points to the next candidate for splitting
- LEVEL contains a value which is the number of times the total space has doubled and is represented by $LEVEL = \lfloor \log_2(n/N) \rfloor$

where n is the current number of buckets and N is the initial number of buckets.

In linear hashing, if the record to be inserted is dynamic, then the mean number of accesses per search approaches 1 and the loading factor is about 90% by using chains according to Litwin [LITW80]. A further advantage of this method is that the address computation is simple and rapid.

Multi-key Hashing Methods

The extendible cell method [TAMM82] is an application of the extendible hashing [FAGI79] to two dimensional files. The methods uses the assumption that the total area $A = [0,1) \times [0,1)$. The point $(x,y) \in A$ has the binary representation

$$x = \sum_{1 \le i < \infty} a_i \, 2^{-i}, \qquad y = \sum_{1 \le i < \infty} b_i \, 2^{-i}.$$

The hashing function, $h(x,y)$, is computed as

$$h(x,y) = \sum_{1 \le i < \infty} (a_i 2^{-(2i-1)} + b_i 2^{-2i}).$$

Here, the extendible cell implementation of a point file F on area A is the structure obtained by applying the extendible hashing method on the region $([0,1))^2$ to $h(F)$. Because calculation of the hash function $h(F)$ can be costly.

However, the extendible cell method has a directory structure. When directory doubling is initiated, the domain is halved alternatively in the x and y direction. However, the directory doubling does not affect the leaf bucket page at all. Directory doubling is initiated by the increment of total depth, d, which is the same as the extendible hashing method. So, instead of using hash function h(F), the two-dimensional directory index function index(x,y) below is used:

$$index(x,y) = 2^{dy} \lfloor 2^{dx} x \rfloor + \lfloor 2^{dy} y \rfloor$$

where $dx = \lceil d/2 \rceil$, $dy = \lfloor d/2 \rfloor$ and total depth $d = dx + dy$

The index function index(x,y) can be used as a two-dimensional directory index. In the case of a structure with more than two-dimensions, the merging process can cause a deadlocked state. To avoid a deadlock, the merging sequence should be made the reverse of the partition sequence. This is effected by maintaining the history of the partition sequence as suggested by Huang [HUAN85].

The storage mapping for multi-dimensional linear dynamic hashing (MLDH) [OUKS83] is adapted from the linear hashing method. As an example of the MLDH method, let l be the total number of partition sequences completed along all dimensions. Because of the cyclic nature of partition sequences along all dimensions, l can be expressed as $l = Ld + r$, where L is the number of splitting cycles completed, and d is a dimension; r is the next bucket for splitting by the given sequence. For example, if the

hashing function H_l is given for the record, R,

where $R = (R_0, R_1, \ldots, R_{d-1})$ for the d dimensions.

$$H_l(R) = (h_{L+1}(R_0), \ldots , h_{L+1}(R_{r-1}), h_L(R_r), \ldots , h_L(R_{d-1}))$$

Then, the next sequence of the split occurs along the axis r and the hashing function H_{l+1} is

$$H_{l+1}(R) = (h_{L+1}(R_0), h_{L+1}(R_1), \ldots , h_{L+1}(R_{r-1}), h_{L+1}(R_r), h_L(R_{r+1}), \ldots , h_L(R_{d-1}))$$

Hence, the sequence of hashing functions H_1, H_2, H_3 is the split function of MLDH if they satisfy the range condition and the split condition below:

RANGE CONDITION

$$H_l : R \rightarrow \{ 0, 1, 2, \ldots , 2^l - 1 \}.$$

SPLIT CONDITION

$$H_{l+1}(R) = H_l(R)$$

or

$$(h_{L+1}(R_0), \ldots , h_{L+1}(R_{r-1}), h_L(R_L + 2^L), h_L(R_{r+1}), \ldots , h_L(R_{d-1}))$$

$$= H_l(R) + (0, 0, 0, \ldots , 2^L, 0, \ldots , 0).$$

For example, insert the two dimensional data below

(10, 201), (45, 300), (77, 2095), (90, 1000),

(100, 103), (4, 4000), (99, 220)

into the MLDH where the bucket capacity b = 2. For the simplicity, no overflow bucket is used.

Initially, the number of the bucket is 1. Records are inserted by $H_0 = (h_0, h_0)$, where $h_0(R) = R \bmod 1$.

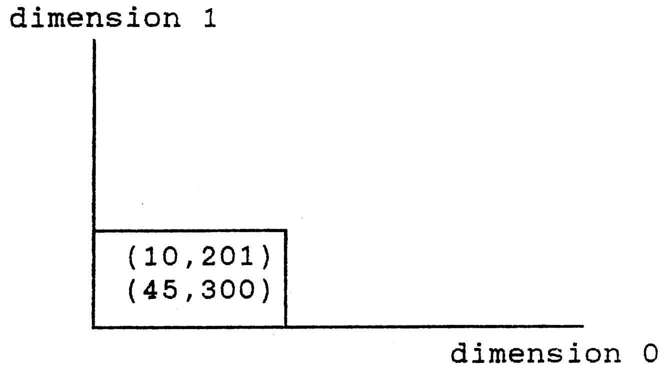


Figure 2-7. $H = (h_0, h_0)$ in a Two-Dimensional MLDH File.

Figure 2-7 depicts the state after two records are inserted. Insertion of record (77, 2095) initiates the splitting of the bucket because the bucket capacity is b = 2.

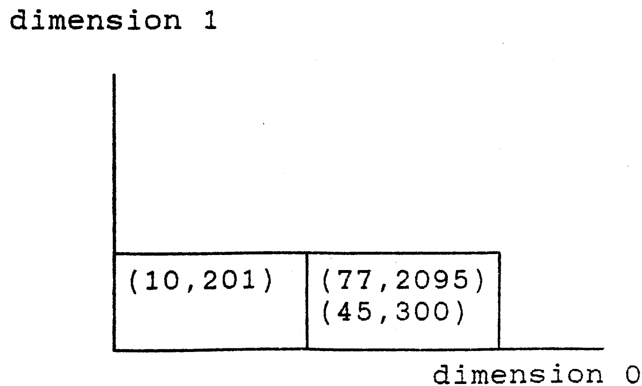


Figure 2-8. $H = (h_1, h_0)$ in a Two-Dimensional MLDH File.

Figure 2-8 depicts the state after one split along dimension 0. The hashing function is

$$H_1 = (h_1, h_0), \text{ where } h_1(R) = R \bmod 2.$$

The continuous insertion of records results in the structure shown in Figure 2-9.

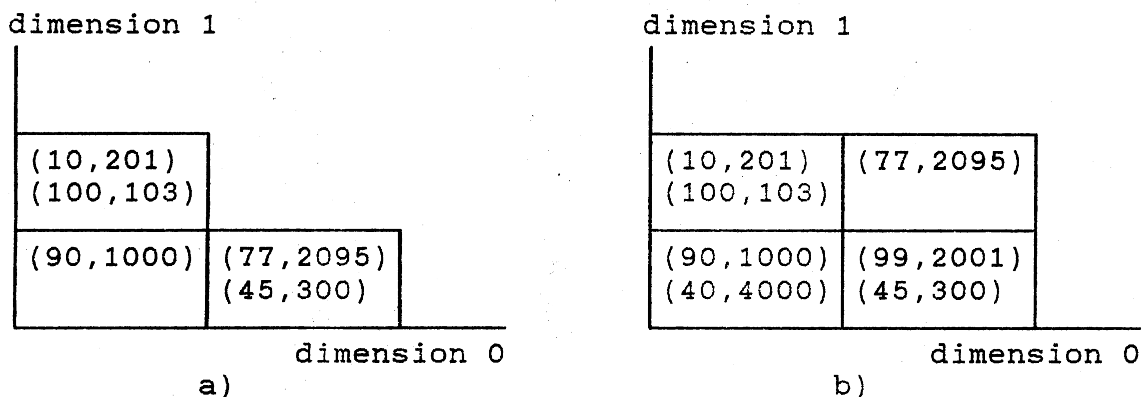


Figure 2-9. $H = (h_1, h_1)$ in a Two-Dimensional MLDH File.

If d is a dimension, and if I^d are d - tuples of integers, then the storage mapping function $M: I^d \rightarrow I$ satisfies

$$M(H_1(R) + (0, 0, .. , 2L, 0, .. , 0)) = M(H_1(R)) + 2^l$$

which follows

$$M(i_0, .., i_{r-1}, i_r + 2^L, i_{r+1}, .., i_{d-1})$$

$$= M(i_0, ..., i_r, ..., i_{d-1}) + 2^l$$

where $l = Ld + r$.

To retrieve a record $R = (R_0, R_1, .., R_{d-1})$ by given

$l = Ld + r$, the following algorithm applies:

1. $H_l(R) = (h_{L+1}(R_0), .., h_{L+1}(R_{r-1}), h_L(R_r), ..., h_L(R_{d-1}))$

2. $idx \leftarrow M(H_l(R)) = \sum_{0 \le i \le d-1} M(P_i(H_l(R)))$

   where P_i determines the coordinate of projection.

3. If idx < NEXT then

   $idx \leftarrow M(H_{l+1}(R))$

The definitions of idx and NEXT are the same as those used for linear hashing.

Compared to the multi-dimensional tree structures [FINK74, ROBI81], the MLDH method does not have any directory structures; therefore, traversing of the directory is not necessary, so the access time of records to approach O(1) from above.

Linear hashing offers several other distinct advantages. An interpolation-based index maintenance [BURK83] is adapted from linear hashing; it does not have a tree-structured directory to traverse. The average successful search length is close to 1, and the space utilization can be easily controllable by the user.[2]

Interpolation-based index maintenance and the MLDH method are derived from linear hashing. Both methods use the address calculation method to find a bucket and use chaining to handle overflow under a controlled loading factor. However, the chaining can cause a long disk access chain for retrieving a record. Both the extendible cell

method and the multidimensional extendible hashing method, derived from extendible hashing, use directory partitioning for handling bucket overflow. However, the partition is not a function of density of data. As a result, if the data is biased heavily over the data area, then the utilization of the directory may be poor [HINR85].

In contrast to other hashing methods, the grid file structure offers significant advantages to streamlining data storage. First, the grid file structure does not have any overflow chains. Second, the partitioning of directory is flexible, since it follows the density of the data distribution. Ultimately, these advantages provide increased flexibility with a maximum two-disk accesses, and adaptability for various data distributions.

---

[2] The theoretical worst case of interpolation-based index maintenance has time complexity O(n) steps for total n records due to allowing the chaining structure to prevent overflow.

CHAPTER III

DATA COMPRESSION METHODS

Many data processing applications store large volumes of alphanumeric data, such as names, addresses, inventory descriptions, or a general ledger account chart description. On-line manuals and help files in computer systems also need massive amounts storage. Recently, the proliferation of computer communication networks and teleprocessing applications further have complicated the questions of adequate storage by involving massive data transfer over long-distance communication links.

Data reduction implies a need to reduce the redundancy in the data representation, that is to compress or compact the data. Since compression techniques reduce the size of a file by 30% - 90% [DAVI73], it is surprising that these technique are not used more widely. The term "data compaction" as defined by Severance [SEVE83], means that the data must be encoded in such a way that data size is reduced while all relevant information is preserved.[1] There are a variety of ways to reduce file sizes.

---

[1] Data compression is a reversible data compaction process which can be used for encryption also.

First, data size can be reduced by avoiding empty space. Empty space avoidance in a file assumes all the fields of the file are a fixed length. If one or more data fields frequently are empty, then the database can be reduced by one additional field. If the field exists, then set its bit to one otherwise, set to zero.

Adaptive pattern substitution is another reduction method. It uses two passes without relying on any existing patterns. An adaptive pattern substitution algorithm first scans the entire data, by examining it for frequent, common patterns of two or more bytes. Then, this method substitutes an unused byte pattern for the frequent common pattern. At that time, the substitution dictionary is updated for that byte pattern. For example, if text is DAABFABCABAB, then this can be replaced as DAαFαCαα where α is the entry α = AB in substitution dictionary. Thus, the total length of this field has been reduced by 4.

For a variety of reasons, commercial databases usually have a large number of blanks, zeroes, and other characters used for the padding empty data or variable-length data fields. The general scheme for suppressing empty fields is called run-length encoding. Martin [MART77] explains a run suppression technique for EBCDIC data with no lowercase characters. Using this method, the second bit of all EBCDIC characters is always 1. When using EBCDIC characters for run-length encoding, the second bit represents the run-length encoding code.

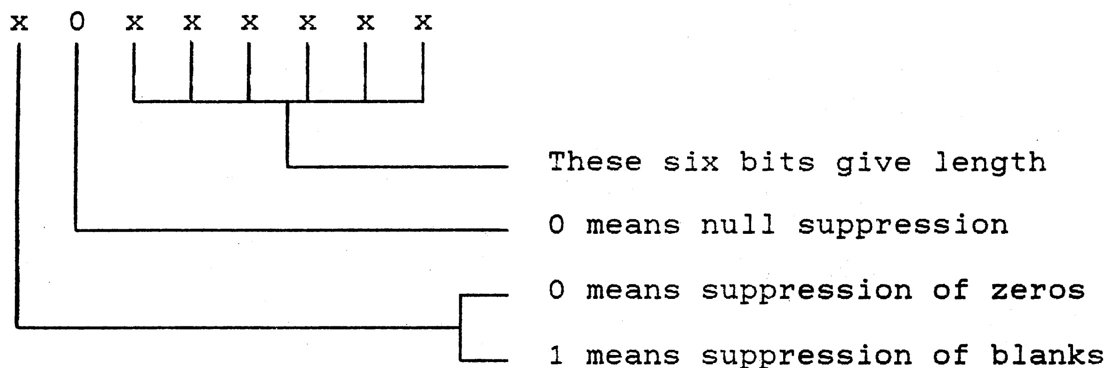


Figure 3-1. Suppression of Repeated Character Method.

For example, if only blanks and zeroes are suppressed as in Figure 3-1, then a single bit can be used to distinguish between them. 6 bits remain to designate run lengths as long as 64 characters. Furthermore, if only a single null type is suppressed, then there is no need to identify it explicitly.

Huffman [HUFF52] also has approached the problem of file-size reduction, developing an optimum method of coding an ensemble of messages consisting of finite numbers. Huffman coding is a variable-length coding which has supplied an optimal length code successfully also. One can construct a Huffman coding for any problem by building a binary tree. Initially, encoding units are listed in order of their probability of occurrence. Next, the two units with the smallest weights are removed from the table: a 0 branch is assigned to one and a 1 branch to the other.

Their weights are added, then assigned to a new combined unit which is merged into the diminished list to maintain order. This procedure is repeated until a single unit remains as the root of the binary tree just as constructed in Figure 3-2.

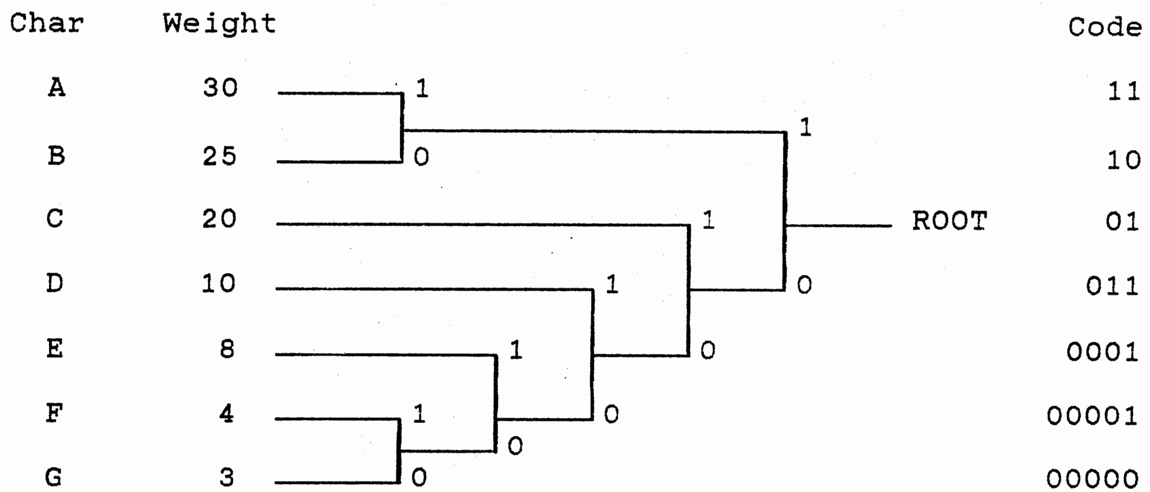


Figure 3-2. Huffman Coding Tree.

The example presented in Figure 3-3 shows the efficiency of this method. The data string in the top portion can be decoded immediately by reading left to right without waiting for the end of the string, since each 0 (or 1) represents the path to the leaf node from the root. The algorithm for implementing encoding and decoding was done by Schwartz [SCHW64].

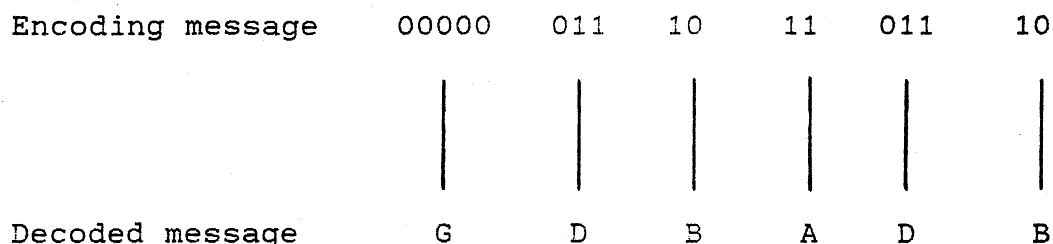


Figure 3-3. Encoding and Decoding of Huffman Coding.

However, if one or more bits are lost during the communication through noise or error, then the data cannot be understood in its entirety. When the third bit of the encoded message in Figure 3-3 has an error which changes it to 1 by any error, then the total message cannot be understood because the Huffman encoding method used a variable length encoding technique.

Even though the Huffman coding method has been evaluated as the optimum minimal encoding method, it presents unique problems: two-pass construction is unavoidable for making the weight table, a factor which can be the weak point of data communication.[2] For example, when the actual weight of each unit is either unknown or changed over time, as is the case with a data message block over a transmission line, then whatever data compression method

---

[2] The weight or the probability of weight already has been studied in commercial database.

is used needs to have only one pass rather than two passes.

A class of universal coding methods [DAVI73, ZIV72, ZIV77, ZIV78] can compensate for this lack of knowledge. By using the sibling property defined by Gallager [GALL78], Knuth [KNUT85] showed that the same weight node in the Huffman tree can be exchanged by the changing of its weight, so the Huffman tree dynamically adapts its structure. Mcintyre [MCIN85] applied the static and dynamic Huffman coding techniques to 530 source programs in four different languages. The results show that the static Huffman coding method makes significantly smaller compressed files than the dynamic Huffman coding method for originally small files. Pechura [PECH82] further surveyed and tested the static Huffman coding method using small computers where the secondary storage is relatively small. The compression potential is invaluable to reclaim the usable storage space. As a further advantage, he also found that both reading and writing speed increased when using compression method.

Despite its obvious advantages, the Huffman code [HUFF52] may have some distinct drawbacks. The variability of the Huffman code may be a drawback because almost all computers are word-oriented rather than bit-oriented. However, this shortcoming may be avoided by using restricted variability codes.

Figure 3-4 presents a conventional 2/4 code where the 11 is used as the switch code. Generally, the 2/4 code can represent 7 characters. In this example, the 3 most

frequently used characters are represented by 2 bits. The switch code "11" is used to distinguish between these frequently used 3 character codes and the less frequently used 4 character codes. However, this restricted variability code generates a less efficient code than the Huffman coding which is explained below. To represent 7 characters, at least 3 bits are needed. Therefore, the total number of bits used for representing characters in Figure 3-4 are 3 x 100 = 300 bits, if the standard character representation method is used.

The restricted variability coding method needs 250 bits, which is as efficient as the Huffman coding (which needs 247 bits in Figure 3-2). If the combined weights of A, B, C are significantly heavier than D, E, F, G, then this

Char	weight	Code
A	30	00
B	25	01
C	20	10
D	10	1100
E	8	1101
F	4	1110
G	3	1111

Figure 3-4. 2/4 Code.

compression method is adequate for the task. Also, by using the fixed length code, data access is ultimately more efficient in performance than the Huffman coding method.

Cormack [CORM85] developed the top-down construction algorithm applied on the IBM's "Information Management System" as yet another method for providing reduction of file size. Although this algorithm is slower than the Huffman coding, Cormack's method is acceptably fast and allows better compression. Additionally, the algorithm does not require a lot of working space for the encoding table.

The Bentry group [BENT86] has suggested a data compression scheme that exploits locality of reference, which occurs when words are used frequently over short intervals in a block and then fall into long periods of disuse. This algorithm is based on a simple heuristic for a self-organizing sequential search. If the message of one block is all upper case letters separated by single spaces and terminated by "!", then for purposes of demonstration we can use

"THE SCOOTER ON THE LEFT HIT THE SCOOTER I LEFT!"

Here, sender and receiver maintain identical arrays using the "move to front" heuristic for fast searching of the frequent word. The arrays are initially empty. To send the word W, the sender searches for the W in the array. If W exists, then the sender transmits I, which is the index of array, then the receiver decodes by writing the Ith element in the array. Then, both move W to the front of their

array, shifting the array from the 1 .. I - 1 to the 2 .. I position. However, if W does not exist in the array of N words, then the sender transmits the integer value N + 1, followed by the word W. The receiver can now detect that the new word should be added to the array if the integer value N + 1 is greater than the size of the current maximum word number N. Then, both sender and receiver move the new word W to the front of their arrays. In this example, after transmitting the first three words of the above message, both of them have the identical array content,

"ON SCOOTER THE"

The next word, "THE", is encoded by the integer 3, because "THE" is already in the array. The entire message is encoded as

1 THE 2 SCOOTER 3 ON 3 4 LEFT 5 HIT 3 5 6 I 5 !

Each word is transmitted as a string of letters exactly once; subsequent occurrences are encoded by integers. The Bentry group [BENT86] reports that this compression method although sometimes significantly better than the word-oriented Huffman coding method is never significantly worse.

By selecting an appropriate method which has a reasonable compression ratio and complexity, compression can be applied to almost all database environments. This is especially true in practical database systems. Decoding time is given the greatest priority because data decoding is required whenever data is retrieved. Encoding, a much less frequently used operation, requires only when insertion or

update is necessary. The only time a large amount of compression is required here is during a complete reloading of the database file.

CHAPTER IV

THE GRID FILE

Introduction

The traditional single key access methods for table look-up on files, such as either the B-tree algorithms or hashing algorithms, rely on having keys with a value within the range of values for the table in the file. However, if the key has many fields or attributes, traditional single key access methods cannot be used. The grid file structure proposed by Nievergelt [NIEV84] is the one of the multikey structures which can handle all attributes without favoring any one attribute. Handling all attributes without favoring any one attribute means treating all attributes as primary keys.

By using scales and directories, the grid file structure performs the two following tasks: the two disk search and efficient range queries with respect to all attributes. An exact matching query must retrieve a single record within at most two disk accesses whether searching is successful or not. If two levels of directory and scales are used for the grid file structure, then the upper level of the structure is composed of the "grid directory" and the "grid scale". They are also called the "resident directory"

and the "resident scale" because they are always kept in the memory. Also, the lower level of structure is composed of the "block directory" and the "block scale", which are also referred to as the "subdirectory" and the "subscale."

The first access to retrieve a record, is to the block directory, which is kept on disk. The result of this access is the loading of the subdirectories and subscales into memory. The second access loads the correct data bucket by the using subscales and subdirectories. The bucket structure must preserve the orders of each of the attribute scales to effect data-key locality.

The grid file structure adapts its shape dynamically from the continuous insertions and deletions of records. The grid file consists of three abstract data types: the linear scale, the directory, and the data bucket. The following illustration explains these three abstract data types.

Let the k-dimensional data area be represented by

$$S = S_0 \times S_1 \times S_2 \times \ldots \times S_{k-1},$$

MaxScale be a maximum number of partitions in one dimension, and r represent the data record. The structures of the grid file can be computed as follows:

LINEAR SCALE

Structure : $Scale_i$[MaxScale] of ScaleType

K one dimensional ScaleType indexed set.

ScaleType has a boundary.

$0 \le i \le k - 1$

Operation :

FindIdx : FindIdx returns the indexes of scales for record r.

PartitionScale : make one partition scale into two.

MergeScale : Merge two scales into one.

DIRECTORY

Structure : $Dir[n_0][n_1] \ldots [n_{k-1}]$ of DirectoryType. one k dimensional DirectoryType set which addresses the data bucket. DirectoryType has a pointer to data bucket and one additional bit to represent shared directory or not.

Operation :

GetDirectory : returns the pointer of the directory

GetBuddyDirectory : returns the pointer of the buddy directory.

PartitionDirectory : partition the directory in two.

MergeDirectory : merge two directories in one.

BUCKET

Structure : fixed-sized type of record, where the actual data records are stored.

Operation :

PartitionBucket : allocate a new bucket storage and separate records in two buckets by the given boundary.

MergeBucket : merge two bucket records into one.

The k linear scales represent k dimensions, i.e., attributes, and are divided by certain intervals using the binary buddy partition method. The k scales are used as the indexes to the directories. Each boundary in a scale represents a (k -1) dimensional partition which separates the directory into two parts. Each grid directory is an one-to-one mapping from the directory to the actual data bucket address in the disk storage. Several grid directories may share one data bucket, but no grid directory can have two or more data buckets. Data buckets are fixed-sized structures of records in which actual data is stored. Using these structures and primitive operations, basic operations, such as find, insert, delete, can be performed.[1]

An example of a multikey searching is shown in Figure 4-1. If the search record is (1980, c), then the search algorithm looks up the scales to find the indexes. By using the indexed directory, the data bucket which holds the searching data can be found easily.

For handling large amounts of data, the grid file has two levels of directories: resident directory and bucket directory. The resident directory always is kept in the memory, while the bucket directory is kept on the disk.

---

[1] The specific structures and operations of the grid file are discussed in the next chapter.

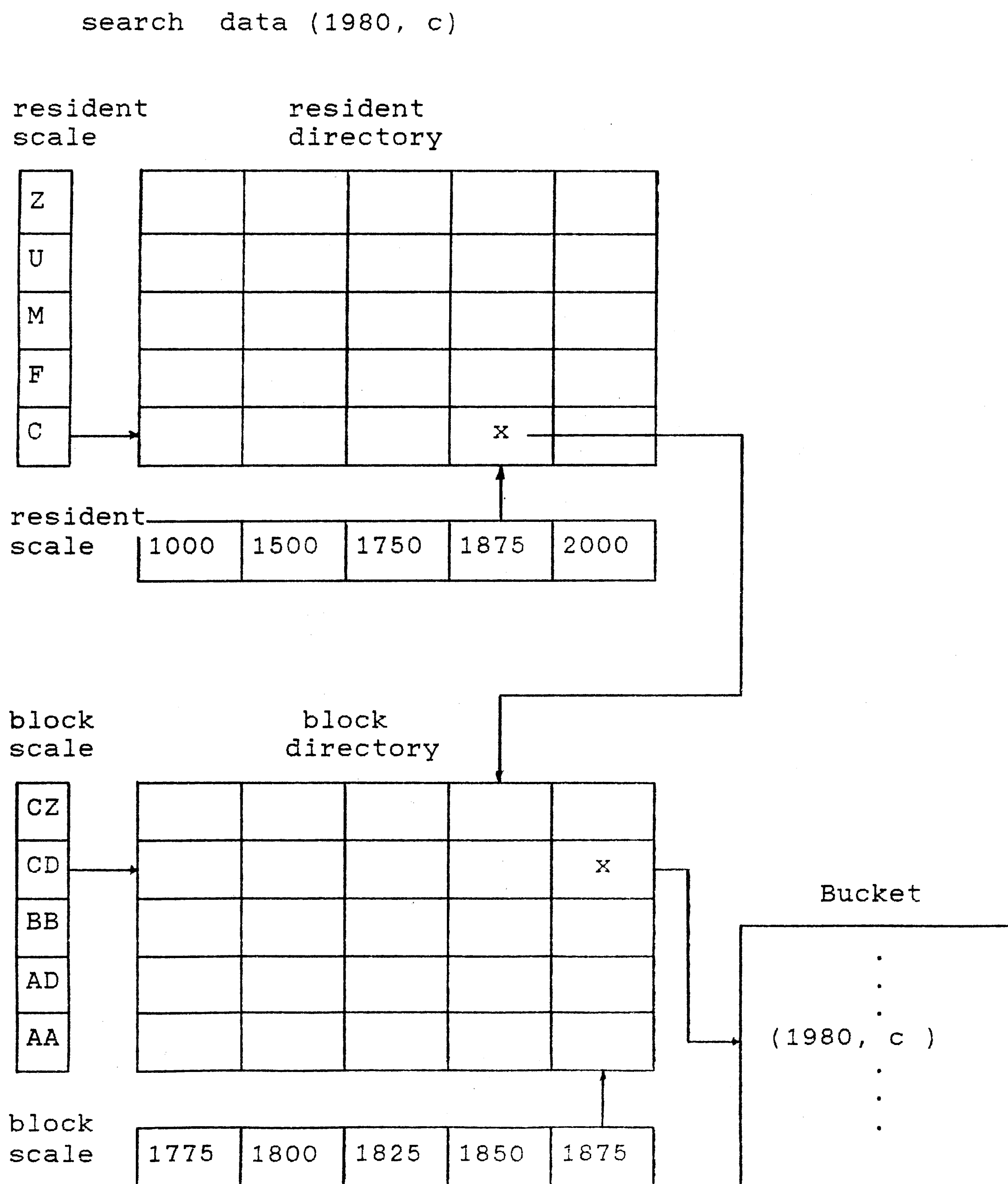


Figure 4-1. A Search for a Record in the Grid File.

A search algorithm references the resident scales and find the appropriate resident directory. The resident directory holds the physical addresses of subdirectories, which are kept on the disk because of large amounts data space they require. After loading the subdirectories and subscales,

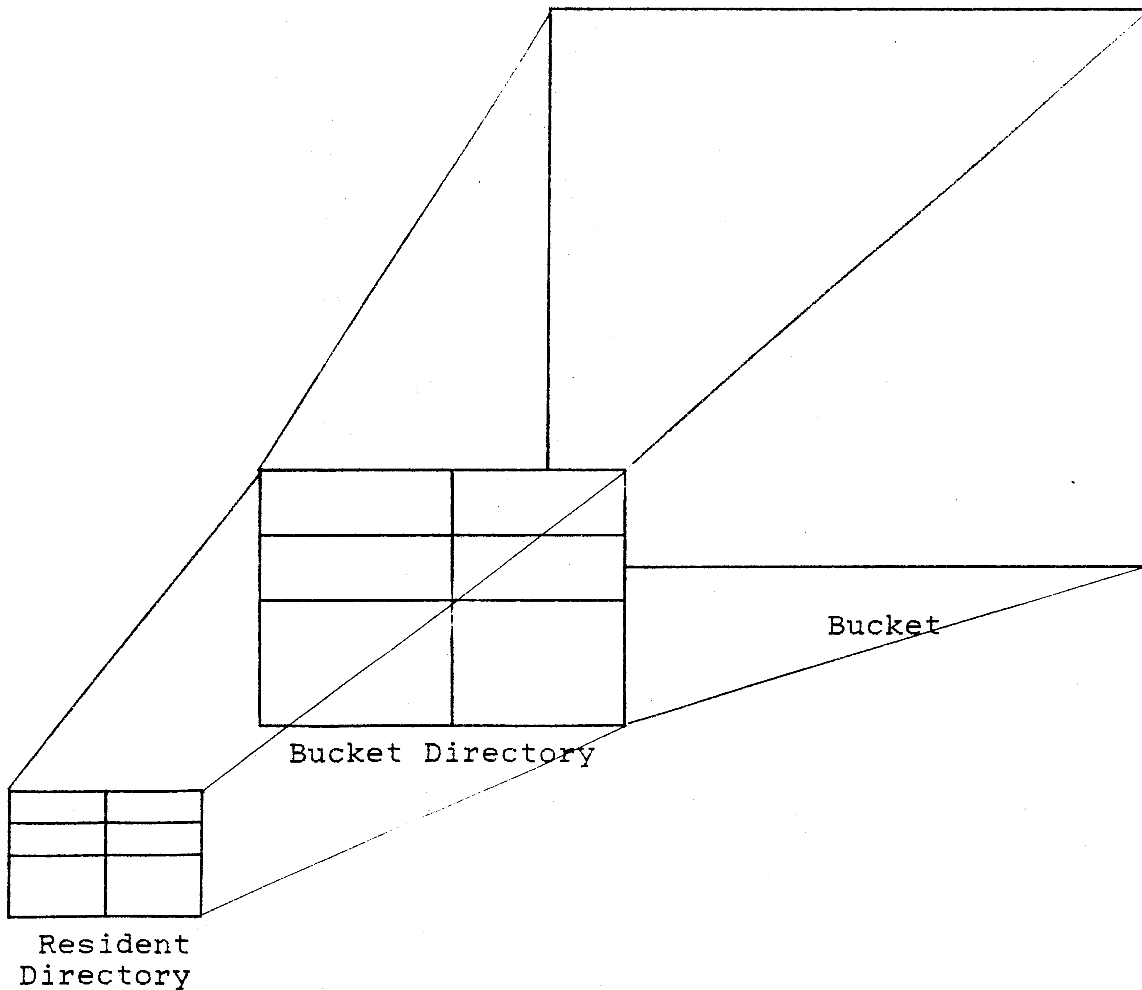


Figure 4-2. Double Levels of the Directory.

the data bucket can be found using the same procedure. As a result, at most two disk accesses are needed for retrieving a record whether it is successful or not.

Adaptability of the Grid File

Figures 4-2 to 4-5 provide an example both of the adaptability and of the dynamics of the grid file. The data bucket is full after continuous insertions are done. Continuous insertion means that the data for insertion is requested continuously. Those requests trigger the partition of the grid directory. As a result, the structure grows by those requests. The merging of two directories is triggered by repeated deletions making the bucket empty.

Initially, a single bucket B1 of capacity c = 4 pictured in Figure 4-3 is assigned to the entire record

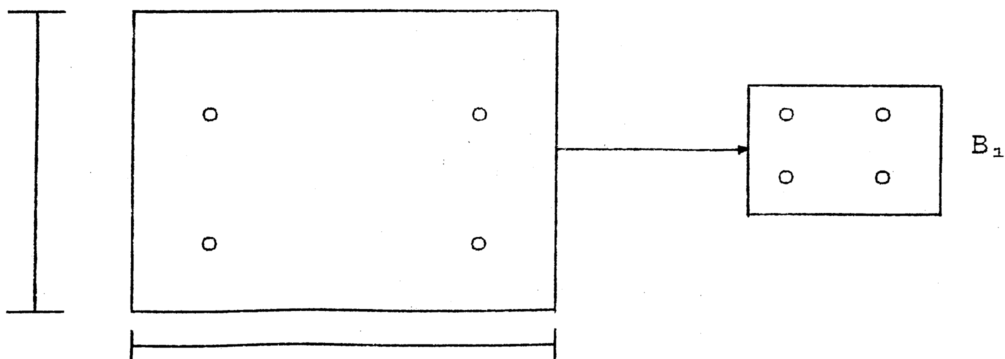


Figure 4-3. Grid Directory and Bucket.

space. Inserting one more record causes the B1 bucket to overflow, resulting in the split of the scale and the directory, and a new bucket B_2 is assigned to the newly

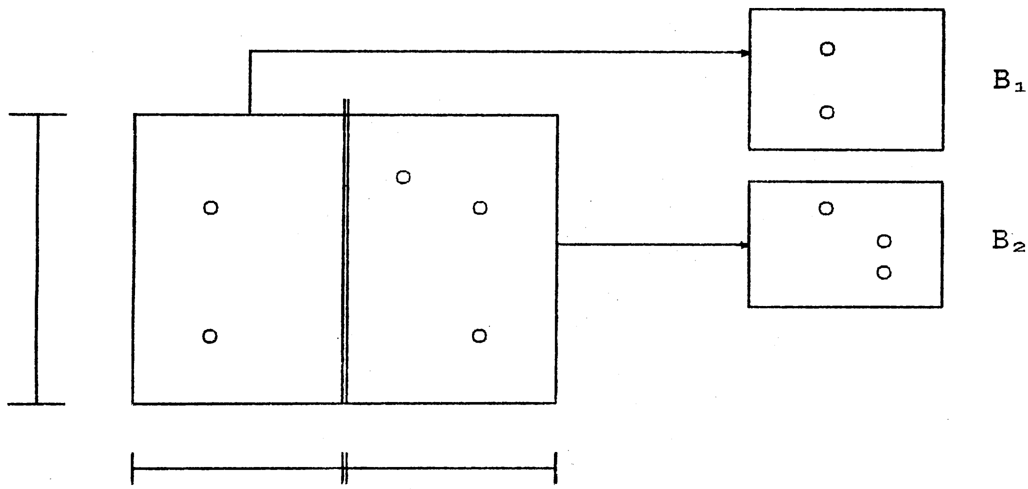


Figure 4-4. Splitting of a Directory.

partitioned directory.

If bucket B_1 overflows again, then its grid directory is split according to the same splitting policy as pictured in Figure 4-4. For the efficient range queries with respect to all attributes, the splitting direction should be alternated. The bottom half of the records which were in B1 are moved to the bucket B_3 as pictured in Figure 4-5. But

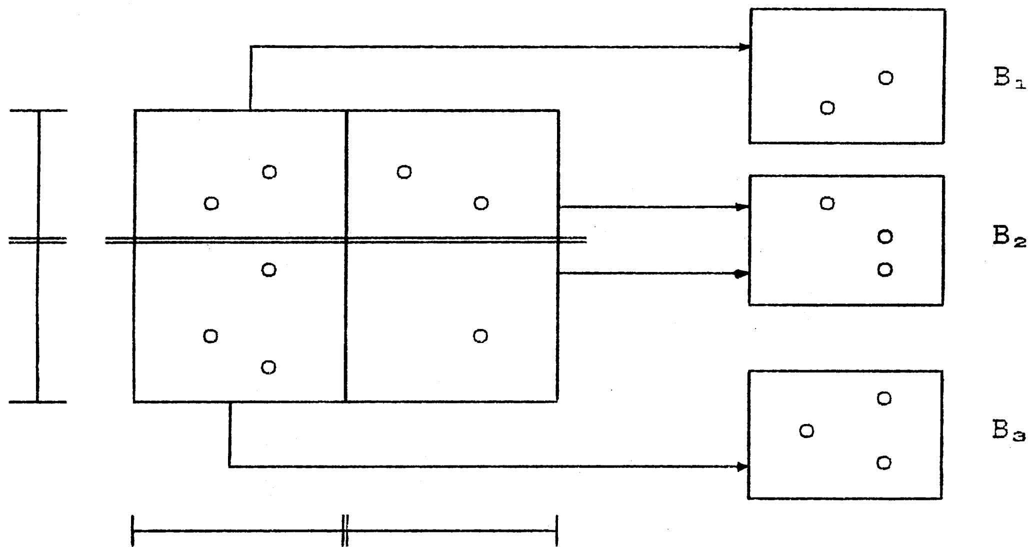


Figure 4-5. Subsequent Splitting of a Directory.

here the bucket B_2 has not overflowed. Consequently, the partitioning of B_2 is not necessary for efficient memory utilization.

Splitting Policy

The grid file structure adapts its shape dynamically to the continuous insertions by splitting its directories and scales. There are several methods for choosing a boundary for a split, including the binary buddy systems [KNUT73, PETE84], the Fibonacci buddy system [BROM80, CRAN75], and the weighted buddy system [CHOW87]. In each case, some range S_n is partitioned into two subranges by the

formula below:

$$S_n = S_{n-1} + S_{n-k}$$

If k = 1, then it is a binary buddy system.

If k = 2, then it is a Fibonacci buddy system.

The grid file structure proposed by Nievergelt [NUEV84] used the binary buddy system. The buddy system is an outgrowth of the neighbor system, both of which are pictured in Figure 4-6. In the k-dimensional buddy system, a merging is performed with exactly one adjacent buddy in each of the k dimensions. In the more generous k-dimensional neighbor system, merging can be performed with either left or right adjacent neighbors. Any of 2k neighbors in k dimensional structure can be merged as illustrated in Figure 4-6. Because, a bucket only can be merged to its buddy to avoid a deadlock situation (defined in the Merging Policy section of this chapter), the buddy system is more restrictive than the neighbor system.

Even though the boundary for each scale can be different types of keys (i.e. integer, character string, etc.), assume that each boundary range is the unit interval [0,1) (left closed and right open) and create a binary buddy system. If an interval [a,b) can be obtained in [0,1) by repeated bisection, then that interval is called a binary radix interval. The number of bisections necessary to get the binary radix interval I is called the level of I and is

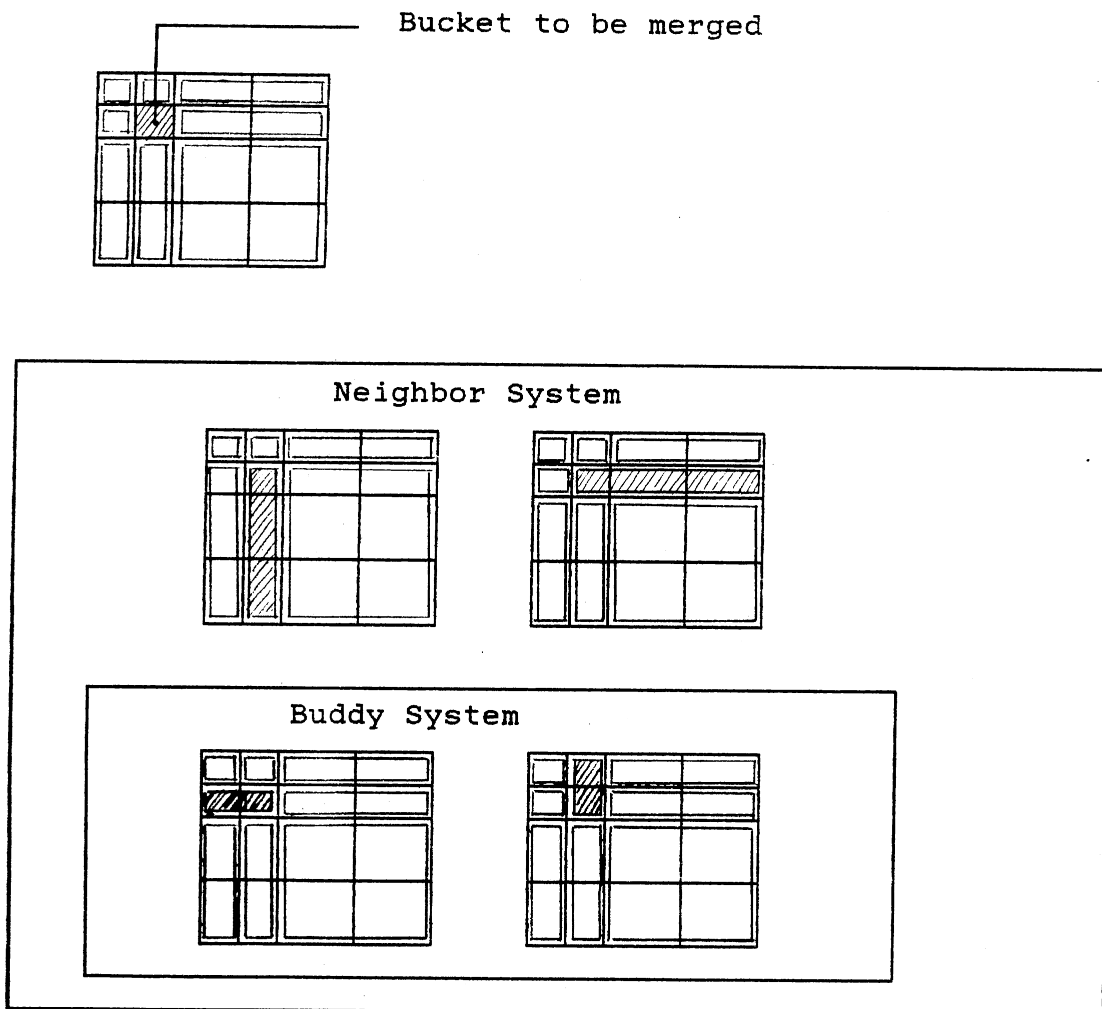


Figure 4-6. Buddy System and Neighbor System.

represented as level(I). Figure 4-7 provides a detailed example of the process. If the interval is [0, ½), then the level is 1 since it requires one bisection of interval [0, 1). For each binary radix interval I which is not [0,1), there is exactly one binary radix interval; i.e. buddy, J such that I and J are disjoint and their union I ∪ J is a

binary radix interval of [0,1).[2]

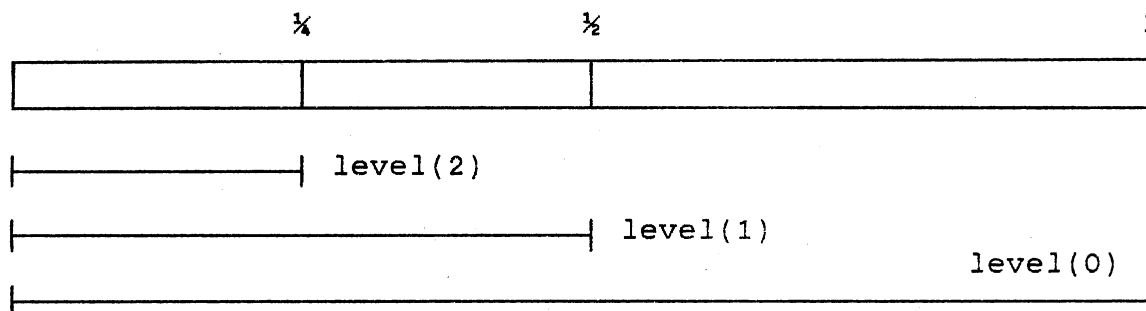


Figure 4-7. Finding Levels of the Scale.

Splitting occurs at four different levels: data buckets; subdirectories and subscales; directory page; root directories and root scales. If a directory page or a data bucket must be split, then one of two things may happen: either several directories share one bucket, or only one directory has the bucket. In the latter case, the boundary should be split into one of the scales and the one-to-one mapping is updated from the directory to the data bucket. In essence, this is the method illustrated in the procession from Figure 4-3 to Figure 4-4.

---

[2] Comparison of the binary buddy system with the Fibonacci buddy system is done by Peterson [PETE84] for the view point of internal and external fragmentation.

Finding the appropriate scale for splitting can be done by the cyclic method or the level method.[3] The cyclic method uses a cyclic sequence for choosing the partitioning scale. For example, if the i_{th} dimensional scale is used for partitioning, then the next candidate for the partition is the $(i + 1)_{th}$ scale. The level method searches the smallest levels of all k dimensional scales and chooses the smallest level scale as the next partition candidate. The new boundary is formed from the bisection of the binary radix interval. From this point, the two methods are identical: the data bucket records are separated according to the scale which is partitioned, and a new bucket is allocated to store the partitioned data records.

Splitting a directory involves an adjustment of scales and directories. The scale is divided into two new scales at the boundary as shown in Figure 4-8. The directory is divided into two parts along the split boundary represented by the double line at the partitioned scale. If the shared directories have to be partitioned, then the buckets assigned to these shared directories cannot be shared. Hence, buckets whose directories are shared at the boundary must be split.

---

[3] The performance difference is analyzed by Han [HAN88].

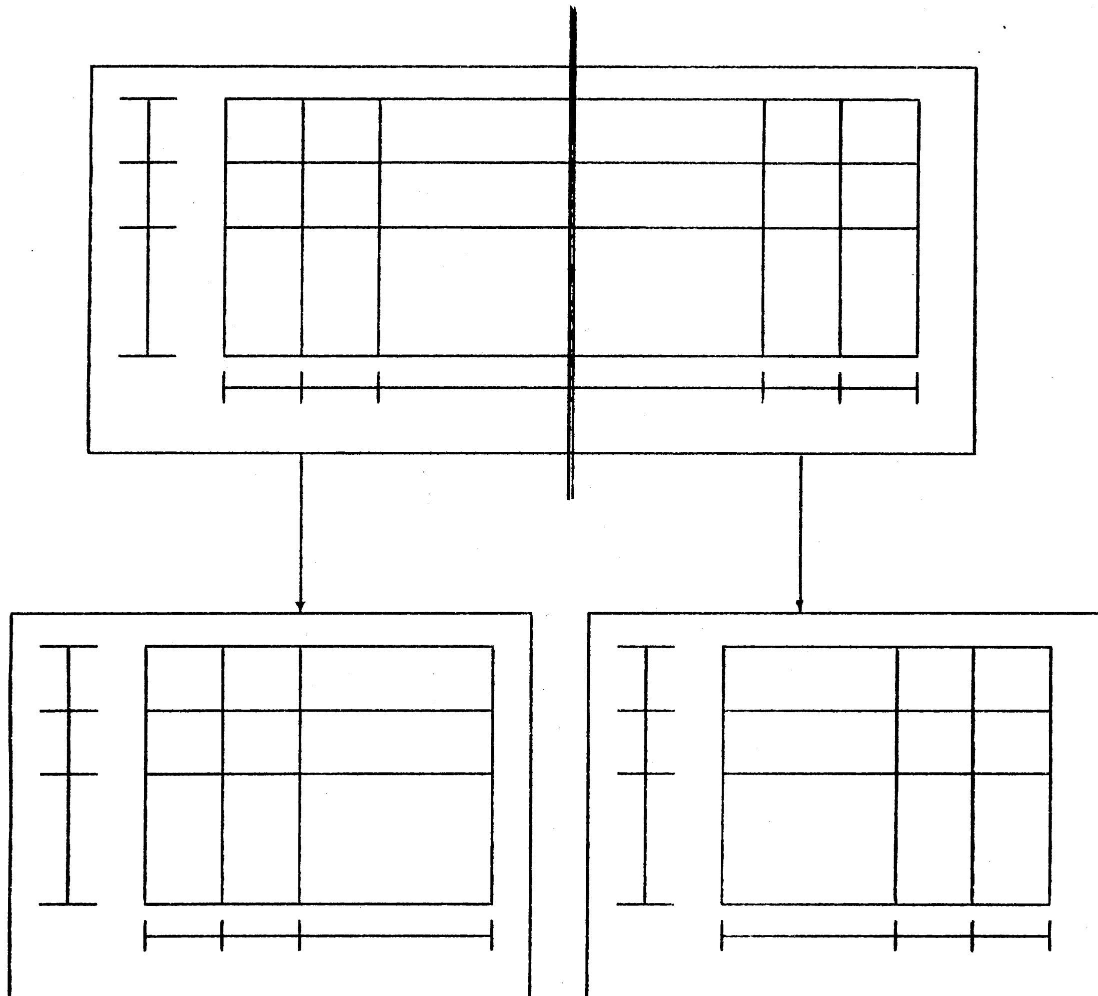


Figure 4-8. Splitting of a Directory Page.

Merging Policy

The grid file merging process is the mirror image of the splitting process. It occurs at all four levels of the splitting process. Although the processing itself is the mirror image of splitting, the merging sequence is not necessarily performed in the reverse order of the splitting sequence, a factor which is derived from inherent

limitations of the extendible cell method [TAMM82]. Similar to the splitting strategy, the candidate normally is limited to its buddy, as long as this buddy is not the neighbor which may cause the deadlock as shown in Figure 4-9.

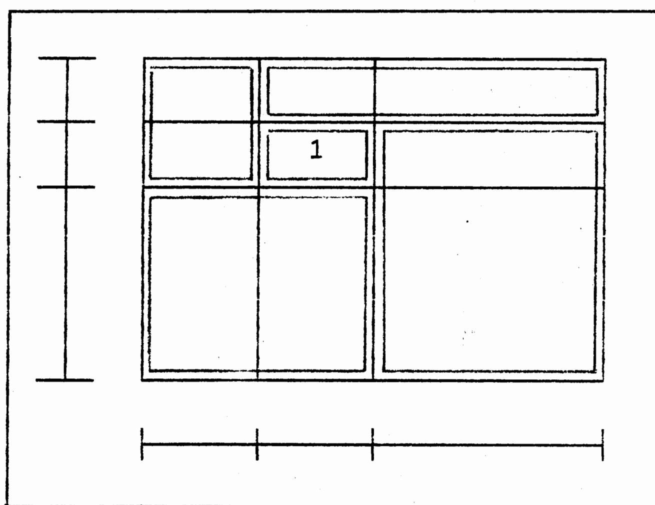


Figure 4-9. Deadlock in a Neighbor System.

Normally, if directory 1 in Figure 4-9 were underflowing, a neighbor directory is sought to initiate merging. However, in this example, there is no directory with which directory 1 can merge. This dilemma illustrates the deadlock problem which is both inherent and problematic in the grid file structure. The buckets cannot be merged again, because they no longer form a box shape. Deadlock is a destructive and significant problem, it leads to a non-

adaptable structure which decreases memory utilization. In the buddy system, every shared directory region is the Cartesian product of binary radix intervals. For this reason, any candidate for merging must be one of the underflowing directory's k dimensional buddies: none of any of 2k buddies of the underflowing directory's neighbor are eligible for merging. Thus, the case in Figure 4-9 cannot occur when the buddy system is used.

With the buddy system, a deadlock situation can also occur in a grid file structure of three or more dimensions.

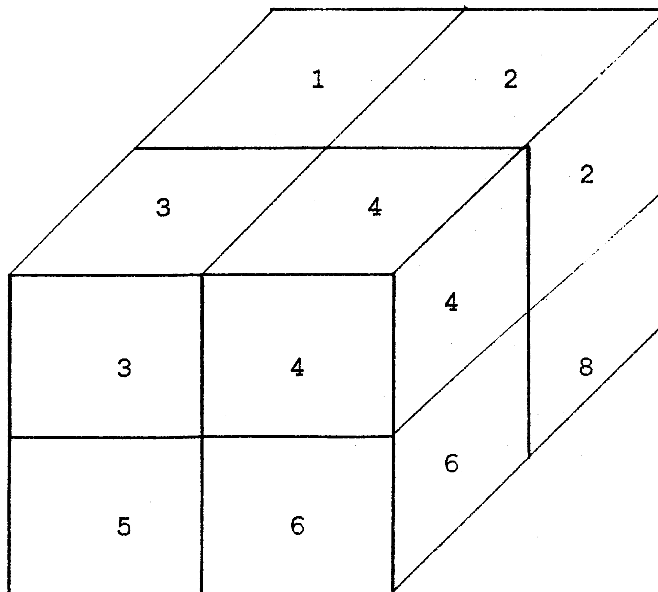


Figure 4-10. Partition of a 3-Dimensional Directory.

In the 3-dimensional grid directory in Figure 4-10, partitioned directories 1 and 2, 6 and 8, and 3 and 5 are merged. The resulting directory shown in Figure 4-11 is in a deadlock state because directory number 4 has no buddy with which to merge.

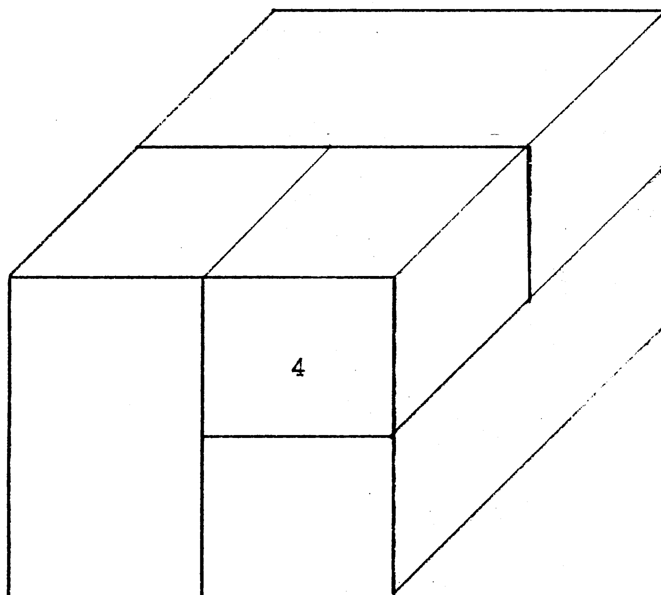


Figure 4-11. A Deadlock State in a 3-dimensional Directory.

This deadlock state can be avoided by an algorithm explained by Hinrich [HINR85]. Hinrich employs two important definitions to explain the deadlock state: the block region and the closed region. The block region is a box-shaped, Cartesian product of binary radix intervals.

The closed region is also box-shaped; here the projection, $\pi_i(k)$, onto each dimension should be a binary radix interval where $0 \le i \le k-1$. Also, if a block region B intersects a closed region C, then all of the directories in the block region B are a subset of closed region C.

To illustrate Hinrich's prevention of deadlock algorithm, B is a block region and C is a closed region. The possibility of deadlock when merging block region D_1 and D_2 can be checked by starting from the total directory T. The particulars of this prevention algorithm are shown in Figure 4-12.

```
D = D₁ ∪ D₂
R = T
MergePossible = True;
while (R <> D) and (MergePossible)
  if exist closed region R₁ and R₂
    where R = R₁ ∪ R₂, R₁ ∩ R₂ = φ, and (D ⊆ R₁ or D ⊆ R₂)
     if (D ⊆ R₁)
       R = R₁
     else R = R₂
  else MergePossible = False
end while
return (MergePossible)
```

Figure 4-12. Deadlock Prevention Algorithm.

A merging algorithm can be developed using Figure 4-12. It is explained in Figure 4-13.

1. Find all k buddies which are potential candidates for merging.
2. Put all candidates in the queue by level.
3. If the queue is not empty, then
    a. get the first candidate in the queue.
    b. Check whether deadlock occurs by using prevention algorithm explained by Figure 4-12.
    c. If deadlock does not occur, then merge the candidate and quit the algorithm.
    d. If a deadlock situation occurs, then repeat step 3.

Figure 4-13. Merging Algorithm.

Buckets need to be merged only when their added occupancy is far below the bucket capacity, which is generally 60% of bucket capacity. Otherwise, a few insertions may produce a splitting operation of the newly merged bucket. Unfortunately, there is no published study about the best threshold for merging buckets. So, the case is remains as an open question. After the buckets are

merged, the scale should be checked as to whether that boundary is still needed or not. If the boundary is no longer necessary, then it is removed and the directories are updated according to the adjusted scale. However, removing the boundary may invoke the directory page merging if its directory utilization is lower than a certain threshold.

Finding the candidate for a merging buddy directory page invokes a situation similar to that illustrated by the merging algorithm. The subscales of the merging dimension can be obtained by appending the two subscales, and is the actual mirror image of the splitting in Figure 4-8. Two directory pages may be merged if the memory utilization of a combined new page is below the threshold of the bucket merging case. The cost of splitting the directory page is significantly higher than that of the bucket merging, because only one disk access is needed. For the boundary which separated the two pages is checked and removed if necessary, which leads to a directory adjustment.

CHAPTER V

COMPRESSION TECHNIQUES APPLIED TO THE GRID FILE STRUCTURE

Directory Compression

The grid file structure proposed by Nievergelt [NIEV84] has two structures available for handling data: a directory and a bucket storage structure. This type of structure allows record retrieval to be accomplished within two disk accesses using the directory in combination with the bucket storage structure. However, partitioning still involves a relatively large number of disk accesses and is somewhat unpredictable because of the distribution factor of the input data.

The level and the buddy of a directory can be calculated by the interval of the directory if a binary buddy system [KNUT73] is used. However, record insertion can be time consuming, as the directory may be partitioned many times before the records in the bucket is partitioned into two buckets. The following example illustrates this problem. Using Figure 5-1, assume bucket capacity to be b = 1. Here, the insertion of two records in the bucket will require four directory partitions because of the

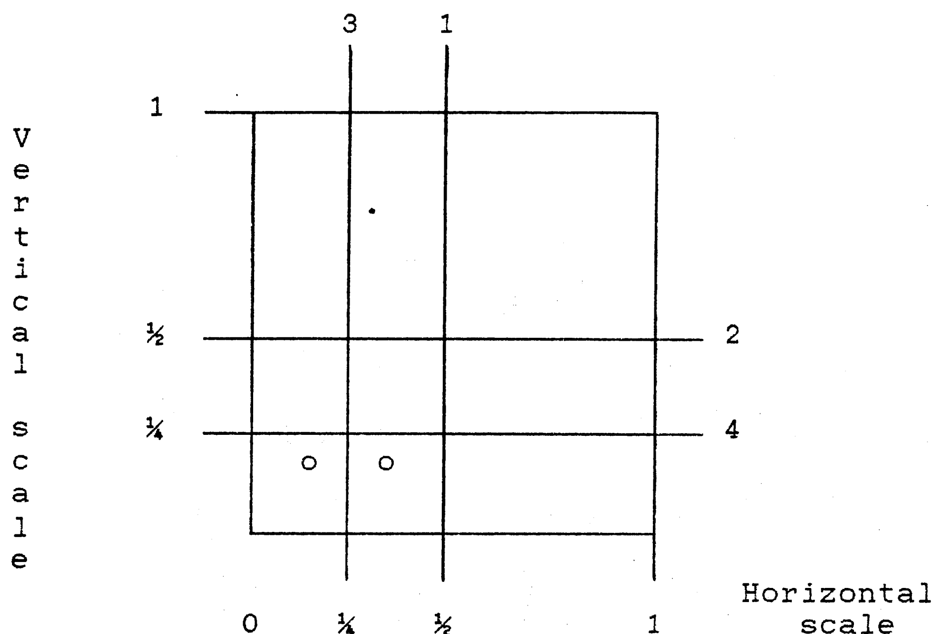


Figure 5-1. 4 Partitions Are Needed in a 2-Dimensional Grid File Structure where Bucket Capacity b = 1.

proximity of the record positions.

Using the extendible cell method [TAMM82] or multidimensional extendible hashing [OUKS83], these structures do have a directory to partition. However, if the partitioned interval is not at exactly the midpoint of the interval, then these structures do not have any way to separate the partitioned directory.

The grid file does, however, have a scale structure which can be used to separate the partitioned directory at some interval other than the exact midpoint. By partitioning the boundary of the scale, an interval value, which is not exactly the midpoint, but which nonetheless guides a subsequent partition of the directory, is assigned.

There is no compelling reason to partition at the midpoint of the interval, especially when data is not well distributed. An alternative to using this interval midpoint or to relying on the binary buddy system is to use a midpoint between given records in a field, hereafter referred to as the "midpoint of record." Figure 5-2 illustrates partitioning using the midpoint of record method.

Figure 5-2 illustrates that, provided the directory and the bucket are partitioned at the midpoint of the input records, any insertion which causes overflow of bucket needs no more than one directory partition.

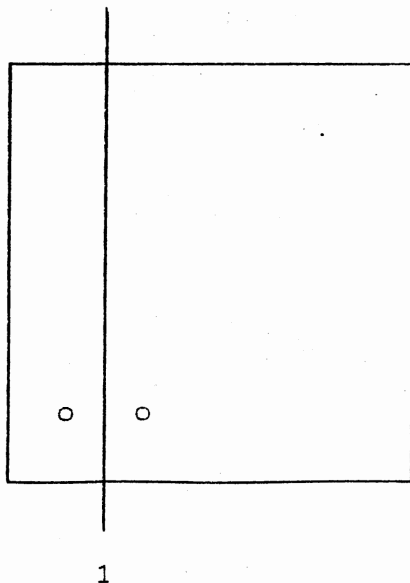


Figure 5-2. Partition at the Midpoint of Record in 2-Dimensional Grid File Structure.

There is a more important reason for using the binary buddy system. Partitioning the space exactly in half, avoids the deadlock state in the grid file structure as defined by Hinrichs [HINR83] and discussed in Chapter IV. If there were another way to find a buddy, then partitioning of the given space in half would not be necessary.

The Cranston method [CRAN75] limits partitioning neither to the space of the interval at midpoint nor to a specific required interval as is normally required by the binary or the Fibonacci method. Cranston developed an easy way to partition and recombine with the Fibonacci buddy system by using two additional bits named the B bit and the M bit. In Figure 5-3, the B bit represents the buddy bit. If the B bit is zero, then the buddy is on its right side, otherwise the buddy is on the left side. The M bit is used to retain data for reconstruction of the original structure.

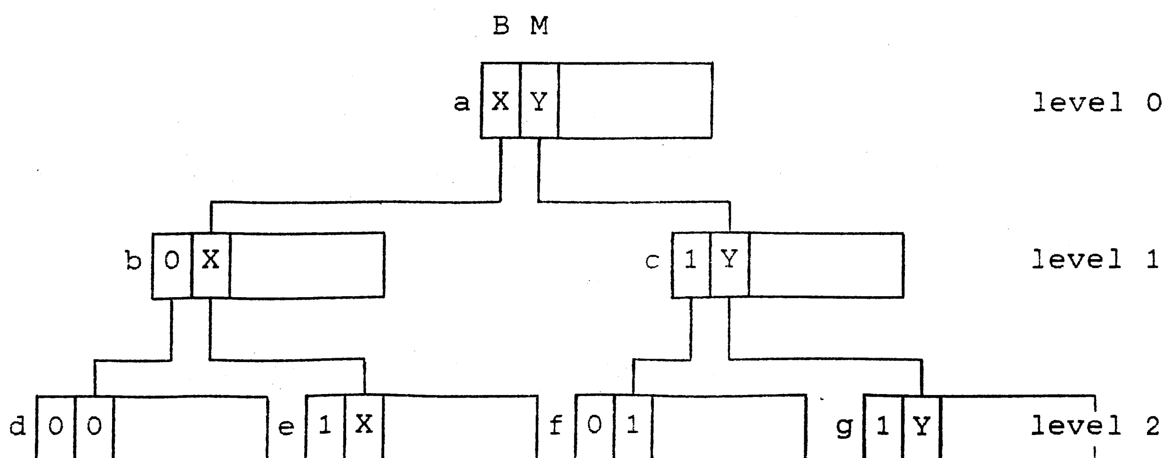


Figure 5-3. Cranston's Partitioning and Merging Method.

Originally, at level 0, the B bit and the M bit are represented as X and Y, where X and Y can be either zero or one. After one partitioning, the level 0 scale is partitioned into two different scales, b and c, at level 1. By the same principle, the level 2 scale is further partitioned from level 1, and thus level 2 has 4 partitions.

The b scale of level 1 has the B bit as zero because its buddy, the c scale, is on the right side. The M bit is X, the B bit of its predecessor scale. Also, the c scale at level 1 has the B bit as one because its buddy, which is the b scale, is on the left side. Here, the M bit is Y, again where Y is the M bit of its predecessor scale. However, in level 2, the e scale, which has B = 1 and M = X, and the f scale, which has B = 0 and M = 1, can not be buddies, but only neighbors. This determination is easily checked by the B bits of both scales. The recombination of scales reverses the order of partitioning. Also, the original X and Y bits can be reconstructed.

Simulations whose data set are a 3-dimensional randomly generated file and the data file for a magazine index illustrate the midpoint of record method . Here, a double-level grid file as defined by Nievergelt [NIEV84] is used for analysis of the directories and scales. However, a single-level grid file result can be extracted by checking the data just before invoking the grid level partition. In the case of this single-level grid file structure, when a randomly generated data set is used, the number of the

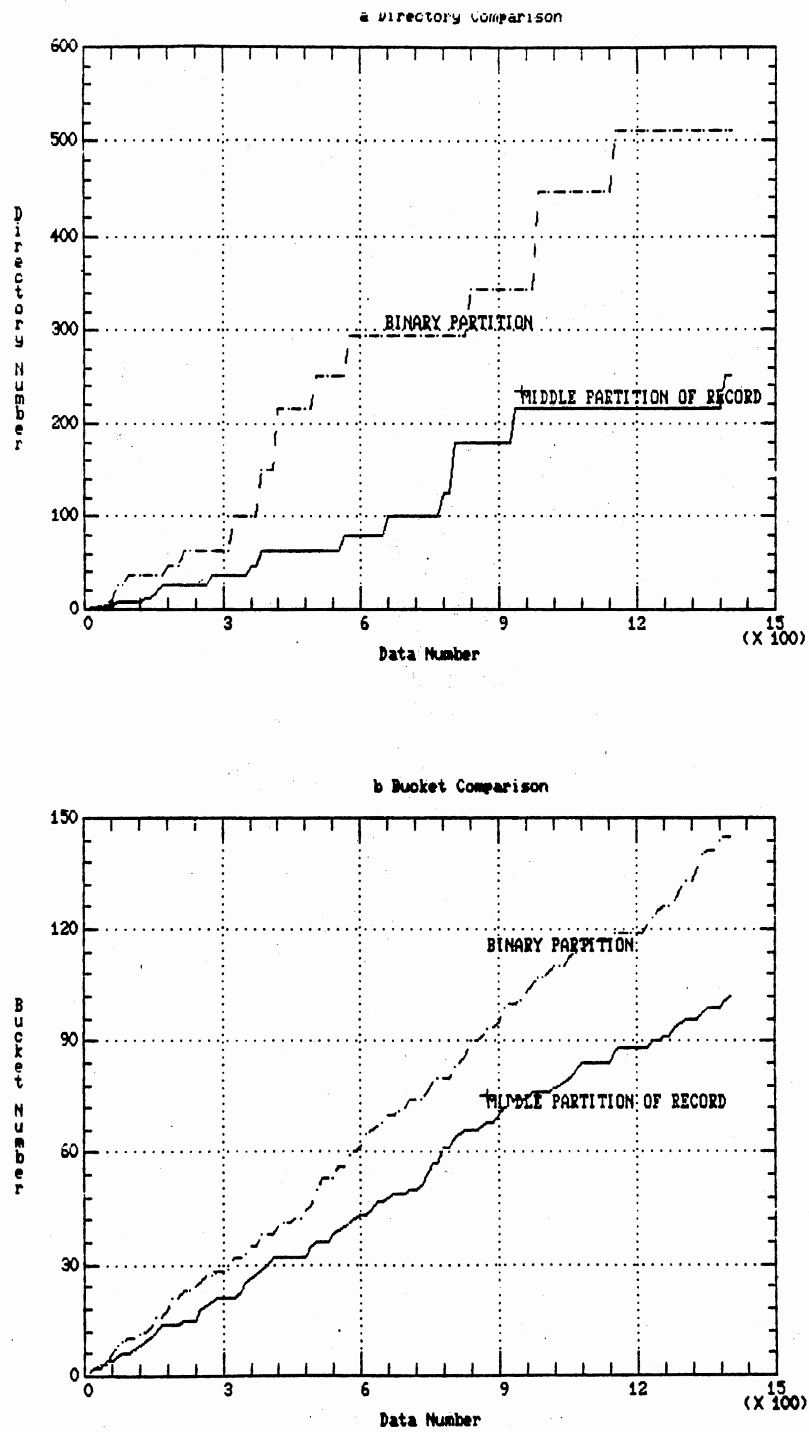
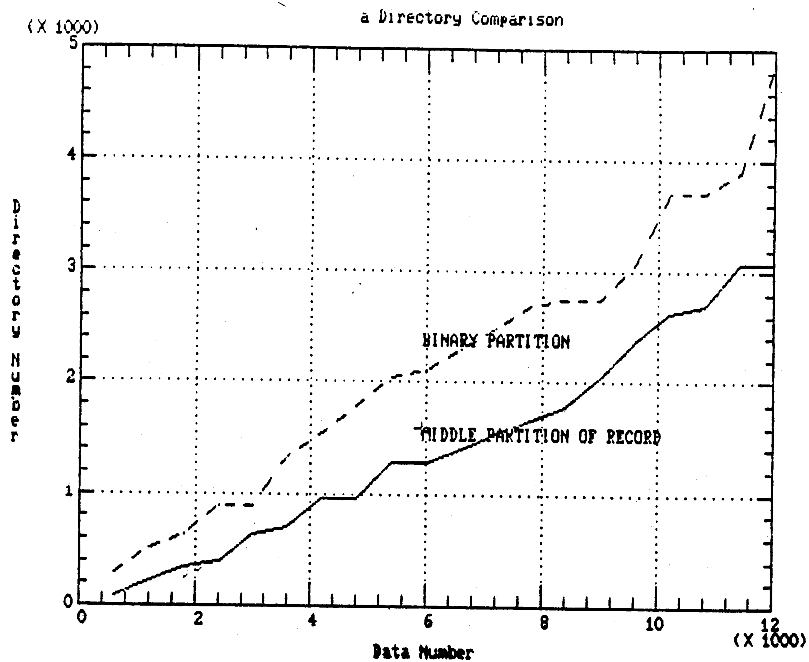


Figure 5-4. Single-Level Biased Data Grid File Comparison.



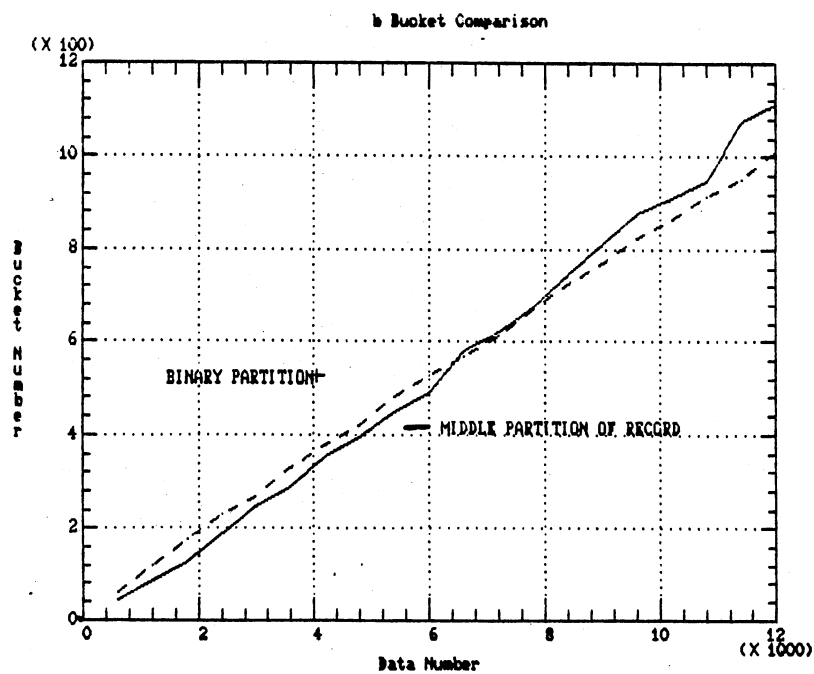


Figure 5-5. Double-Level Biased Data Grid File Comparison.

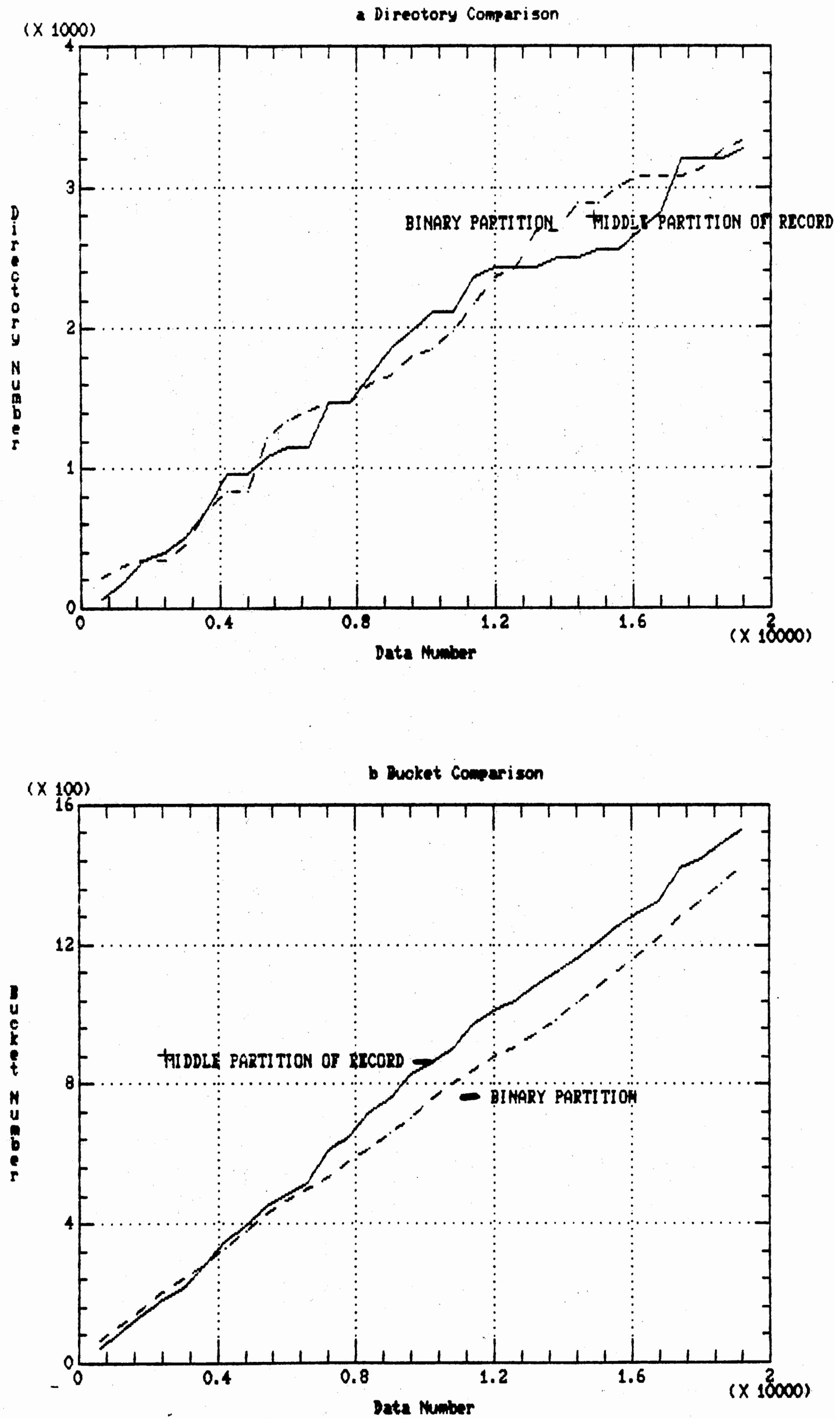


Figure 5-6. Double-Level Random Data Grid File Comparison.

directories and the number of the buckets are very close in both the binary buddy and midpoint of record methods even though the method of partitioning is different. However, as illustrated by Figure 5-4.a, when a biased data set is used, the number of the directories in the midpoint of record partition method is better than the binary partition method by a factor of two. Biased data means that the data set is not distributed equally over the total data area. If the biased level is heavier, then the number of directories may be reduced further.

In the case where two levels of the grid file with a biased data file are used, then the number of directories in the midpoint of record method is reduced to almost 65% of the binary method, as shown in Figure 5-5.a. However, the number of buckets required is slightly more than that of the binary method. Also, as Figure 5-6 shows, the number of directories in the midpoint of record method is slightly better, while the number of buckets is slightly worse. The reason for this result is that the upper, grid level partitioning cannot be done by using the interval midpoint for the middle block scale value, because each block directory and the block scales have different boundaries for the different sets of data. In the simulation, grid level partitioning is done by the binary method. As a result, each grid directory partition induces the bucket partitions of $n - 1$ hyperplanes of its subdirectories. For example, originally, the bucket capacity was $b = 1$ and the maximum

number of partitions was 2 in Figure 5-7.a. Here, the left upper corner numbers represent the indexes of the directories. If a new record x in bucket number 3 is

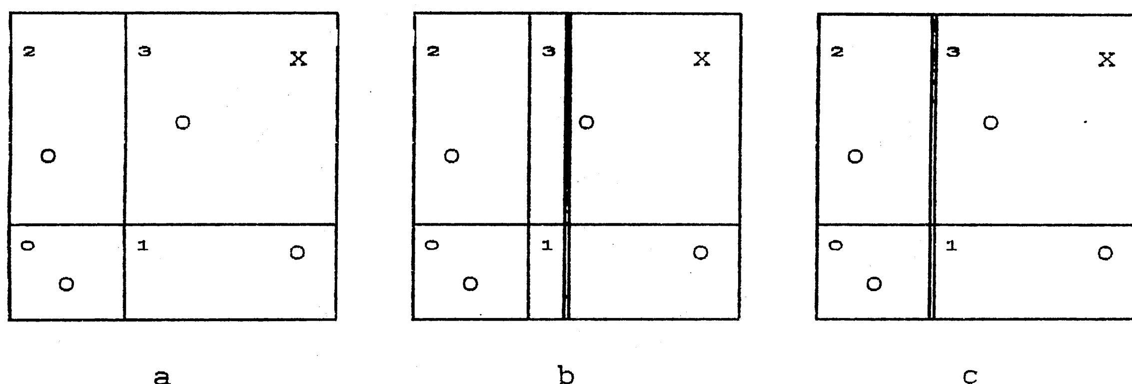


Figure 5-7. Grid Partition Methods.

inserted, a grid partition, represented by the double line in Figure 5-7, is necessary. Block partition is impossible since the maximum number of partitions for the block limited to only 2.

In Figure 5-7.b, a binary partition occurs, while in Figure 5-7.c the midpoint of record method is used. In the case of Figure 5-7.b, the records in the bucket pointed by directory 1 and directory 3 must be partitioned because different block directory and block scales are unable to share a single bucket while maintaining the two-disk access principle. Even though the midpoint of record partition

method is used for the grid level partition, all block directories in the n - 1 hyperplane, except the current block directories, should separate their bucket records for the same reason as indicated in Figure 5-7.b. This is the main reason that the bucket numbers of the midpoint of record partition are more than those of the binary partition method in Figure 5-5.b and Figure 5-6.b.

There is one other advantage to the midpoint of record partition method. Because the partitioning follows the midpoint of the block directory, it is guaranteed that one grid partition and one block partition are enough to divide any kind of the data set for a partition. Thus, the worst case number disk accesses in the partition is fixed to the n - 1 hyperplanes of both grid partition and block partitions. The abstract data type and its operations of the partition by the midpoint of record is:

ABSTRACT DATA TYPE FOR A SCALE

Scale [MaxScale] of ScaleType

$1 < MaxScale < \infty$, MaxScale : integer

Structure of ScaleType

B bit : Buddy bit, if buddy is right side then 0 else 1

M bit : Memory bit, used for reconstructing original memory

UpperBound : biggest data fit into this memory

Operation :

Partition(Scale, i, NewBoudnary) :

Partition of Scale [i] into Scale [i] and Scale [i+1].

Scale[i+1].UpperBound = Scale[i].UpperBound

Scale[i+1].B = 1

Scale [i+1].M = Scale[i].M

Scale[i].UpperBound = NewBoundary

Scale [i].M = Scale [i].B

Scale [i].B = 0

Merging(Scale, i) : Merging of Scale [i] and Scale[i+1] into Scale [i]

Scale [i].B = Scale [i].M

Scale [i].M = Scale [i+1].M

Scale [i].UpperBound = Scale [i+1].UpperBound

Bucket Compression

The bucket storage is a fixed sized structure; its occupancy is about 69% [NIEV84] if a relatively large bucket size is used. Also, each field does not consume all the space it has available for storage (i.e. internal fragmentation of field). Thus, by its very nature, this structure presents the possibility of compression due to this habitually unused space. The actual file size, therefore, may reduced significantly if, instead of using fixed-length record bucket size, a variable-length record bucket is used. However, managing the variable length file is not easy.

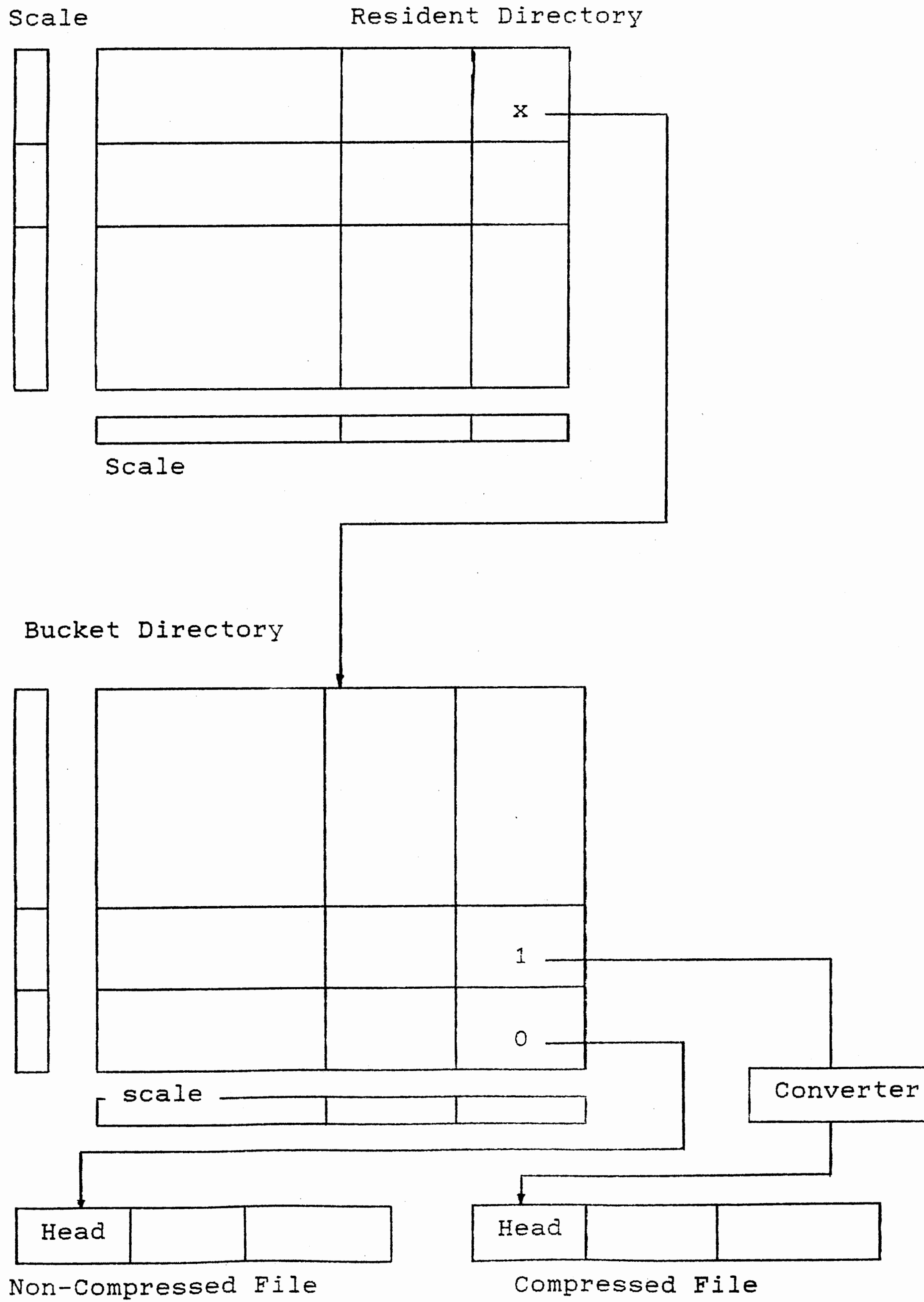


Figure 5-8. Compression of Bucket Record.

An alternative to using a variable length bucket size is to apply two different bucket storages and to try to reduce the total size of bucket file. This alternative is presented in Figure 5-8. Here, two different bucket storage structures are created and a reduction method is applied to try to compress the total size of the bucket file. In this structure, the Huffman compression technique [HUFF52] is used for compressing bucket records. One of the bucket structures has the original size and stores records without using any compression technique. The other one has a smaller sized compressed bucket and stores records in a compressed form. In this case, the main reason for using two different sized buckets is to overcome the worst-case of the Huffman method, which is explained in Chapter III. Obviously, the total file size never outgrows the grid file bucket size.

A further aid in reducing the complexity of the compression technique may be to establish a certain threshold for compression. For example, if bucket occupancy is less than 70%, then compress the bucket records, otherwise leave them in the non-compressed form.

The converter of the grid file in Figure 5-4 is composed into a Compressor and Decompressor. The abstract data type of Compressor is:

Huffman tree[1] of HuffmanNodeType

Structure of the HuffmanNodeType:

char : character of the node if the node is external.

left : pointer to the left son

right : pointer to the right son

father : pointer to the father pointer

Operation :

FindCharater (BitString, char)

search Huffman tree by given bit-string until the external node.

return the character of that external node.

The abstract data type of the Decompressor is also presented:

Huffman table [NumChar] of HuffmanTableType.

$0 < NumChar < \infty$, NumChar : integer.

Structure of the HuffmanTableType

char: character for translating into bit-string

BitString: bit-string of the character form

Operation :

FindBitString(char, BitString)

Search for the character in the Huffman table.

Return the BitString.

---

[1] The construction of the Huffman tree algorithm was developed by Schwartz [SCHW64].

By changing the compressed bucket size from zero to the same size as the bucket size, the total file size can be easily searched. Figure 5-9 illustrates this concept by using the binary partition method. Here the bucket size is 10 records and the 3000 records are being inserted. As Figure 5-9 shows, the graph experiences an almost 66% reduction of the original file size.

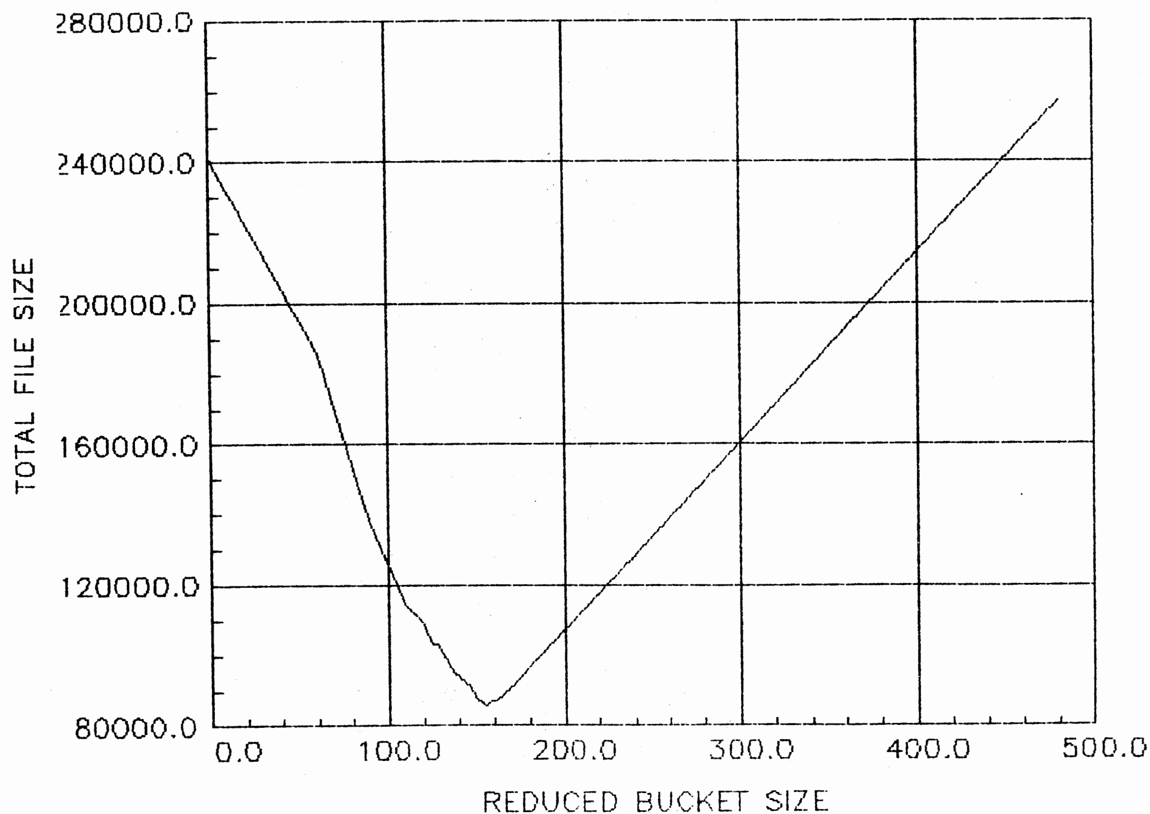


Figure 5-9. Total File Size Variation Using a Biased Data File.

In summary, the midpoint of record partition method has several characteristic results. In the case of a single-level grid file structure, if the midpoint of record partition method is used, the compression ratio of bucket and directory is proportional to the distributions of the incoming data. If a double-level grid file structure is used, the bucket storage is increased if the midpoint of record partition method is used. However, the directory size is reduced to around 65% of the original directory size, and it is also proportional to the distribution of the data. The bucket reduction ratio in Appendix B further illustrates the compression ratio's relation to bucket occupancy.

CHAPTER VI

SUMMARY, CONCLUSION, AND SUGGESTED FUTURE WORK

In conclusion, there are several reasons for adapting the midpoint of record method. The simulation clearly demonstrates that this method has several advantages over the binary buddy system. The midpoint of record method limits worst case accesses for the partition. Furthermore, in the case of a single-level grid file structure for a biased data set, compression is enhanced by a factor of two for the directory, while the number of buckets is reduced to nearly 70% of the binary buddy system. Similarly, in the double-level grid file structure for a biased data set, the directories are compressed to 65% of their original size. An extrapolation from experience with the biased data set indicates the results should be similar for a random data set. Further, if the data file is much more heavily biased than the files of our simulation, then the compression ratio should be better than the factor of two compression results of this study. Thus, the results reveal that, despite claims for the dynamic nature of a grid file structure that utilizes the binary buddy system, the new partitioning method, the midpoint of record method, provides a significantly more dynamic adaptation for an incoming data

set.

In the case of the bucket storage, the bucket compressions of the grid file structure yields compression ratio of 66% mentioned in the Chapter V. Also, even when a randomly generated file is used as a data file, the compression ratio is around 59% due to the internal fragmentation of the input file and the two different bucket sizes. (See the graph in Figure 6-1.)

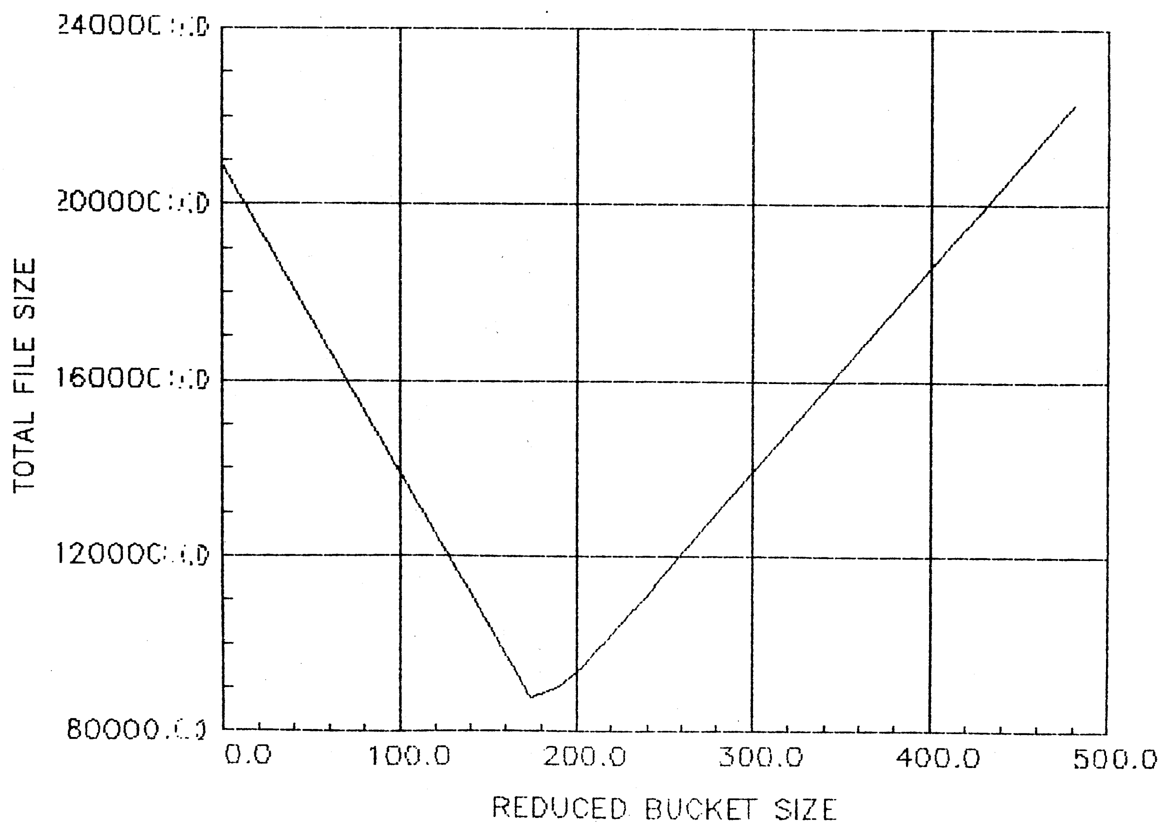


Figure 6-1. Total File Size Variation using a Randomly Generated Data File.

There are several directions that need to be explored in the future using this study as a base. The binary partitioning method and the midpoint of record method ought to be compared in the upper level of the grid file to see if the results are comparable to those examined in the lower level by this study. This research may add to the efficiency of the midpoint of record for it yields an even more dramatic reduction in the overall numbers of directories. More study must also be applied to the relationship between bucket occupancy and optimal reduced bucket size. This information may lead to a chart showing optimal reduced bucket size for a variety of data files. Other users could avoid costly and time-consuming calculation of optimal reduced bucket size by consulting this chart.

BIBLIOGRAPHY

[BENT86] Bentley, J. L., and Sleator, D. D., and Tarjan, R. E., and Wei, V. K. "A locally adaptive Data Compression Scheme." CACM 29, 4 (Apr. 1986), 320-330.

[BROM80] Bromley, A. G. "Memory fragmentation in buddy methods for dynamic storage allocation." Acta Inf. 14 (1980), 107-117.

[BURK83] Burkhard, W. A. "Interpolation-based index maintenance." BIT 23 (1983), 274-294.

[CHOW87] Chowdhury, S. K., and Srimani, P. K. "Worst case performance of weighted buddy systems." Acta Inf. 24 (1987), 555-564.

[CORM85] Cormack, G. V. "Data compression on a database system." CACM 28, 12 (Dec. 1985), 1336-1342.

[CRAN75] Cranston, B., and Thomas, R. "A simplified recombination scheme for the Fibonacci buddy system." CACM 18, 6 (June 1975), 331-332.

[DAVI73] Davisson, L. D. "Universal noiseless coding." IEEE Trans. Inf. Theory 19 (1973), 783-795.

[FAGI79] Fagin, R., and Nievergelt, J., and Pippenger, N., and Strong, H.R. "Extendible hashing-A fast access method for dynamic files." ACM Trans. Database Syst. 4, 3 (Sept. 1979), 315-344.

[FINK74] Finkel, R. A., and Bentley, J. L. "Quad trees-a data structure for retrieval on composite keys." Acta Inf. 4 (1974), 1-9.

[FRED60] Fredkin, E. "Trie memory." Comm. ACM 3, 9 (Sept. 1960), 490-499.

[GALL78] Gallager, R. G. "Variations on a theme by Huffman." IEEE Trans. Inf. Theory IT-24 (1978), 668-674.

[HAN88] Han, C. "A grid file approach to large multidimensional dynamic data structures." MS Thesis, Oklahoma State University, (May 1988).

[HINR83] Hinrichs, K., and Nievergelt, J. "The grid file: a data structure designed to support proximity queries on spatial objects." In Proc. Workshop on Graph Theoretic Concepts in Computer Science, Osnabruck, (1983), 100-113.

[HINR85] Hinrichs, K. "Implementation of the grid file: design concepts and experience." BIT 25, 4 (1985), 569-592.

[HORO84] Horowitz, E., and Ai-Suwaiyel, M. "Algorithms for Trie Compaction." ACM Trans. Database Systems, 9, 2 (June 1984), 243-263.

[HUAN85] Huang, S. "Multidimensional Extendible Hashing for Partial-Match Queries." Int JC Inf 14, 2 (1985), 73-82.

[HUFF52] Huffman, D. "A method for the construction of minimum redundancy codes." Proc. I. R. E. 40, 9 (Sept. 1952), 1098-1101.

[KNUT73] Knuth, D. E. "the Art of Computer Programming." Vol. 3. Addison-Wesley, Reading, Mass., (1973).

[KNUT85] Knuth. G. E. "Dynamic Huffman Coding." J. Algorithms 6, 2 (June 1985), 163-180.

[LARS87] Larson, P. E. "Dynamic hashing" BIT 18 (1987), 184-201.

[LITW80] Litwin, W. "Linear Hashing: a new tool for file and table addressing." In Proc. 6th International Conf. on Very Large Data Bases, (1980), 212-223.

[MART77] Martin, J. "Computer Database Organization." Englewood Cliffs. N.J. Printice-Hall, (1977).

[MCIN85] Mcintyre, D. R., and Pechura, M. A. "Data compression using static Huffman code-decode tables." CACM 28, 6 (June 1985), 612-616.

[NIEV84] Nievergelt, J., and Hinterberger, H., and Sevcik, K. "The Grid File: An adaptable, symmetric multikey file structure." ACM Trans. Database Systems, 9, 1 (Mar. 1984), 38-71.

[OUKS83] Ouksel, M., and Scheuermann, P. "Storage mapping for multidimensional linear dynamic hashing." Proc. of the Second ACM SIGACT-SIGMOD symposium, (Mar. 1983) 90-105.

[PECH82] Pechura, M. "File archival techniques using data compression." CACM 25, 9 (Sept. 1982), 605-609.

[PETE84] Peterson, J. L., and Norman T. A. "Buddy systems." ACM Trans. Database Systems, 9, 1 (Mar. 1984), 89-99.

[RAMA85] Ramamohanarao, K., and Sacks, R. "Partial match retrieval using recursive linear hashing." BIT (1985), 477-484.

[REGN85] Regnier, U. "Analysis of grid file algorithm.", BIT 25 (1985), 335-357.

[ROBI81] Robinson, J. "The K-D-B tree: A search for large multidimensional dynamic Indexes." Proc. ACM SIGMOD (1981), 10-18.

[SALZ86] Salzberg, B. "Grid file concurrency." Inf. Syst. 11, 3 (1986), 235-244.

[SARI87] Saritepe, H. N. A. "An analytic comparison of grid file and k-d-b tree structures." MS Thesis, Oklahoma State University, (Dec. 1987).

[SCHW64] Schwartz, E. S., and Kallick, B. "Generating a canonical prefix encoding." CACM 7, 3 (Mar. 1964), 166-169.

[SEVE83] Severance, D. "A practitioner's guide to data base compression." Inf. Syst. 8, 1 (1983), 51-62.

[TAMM82] Tamminen, M. "The extendible cell method for closest point problems." BIT 22 (1982), 27-41.

[VALL76] Vallarino, O. "On the use of bit maps for multiple key retrieval." ACM SIGPLAN Notices 11 (Mar. 1976), 108-114.

[ZIV72] Ziv, J. "Coding of sources with unknown statics - part I. Probability of encoding error." IEEE Trans. Inf. Theory (1972), 384-394.

[ZIV77] Ziv, J., and Lempel, A. "A universal algorithm for sequential data compression." IEEE Trans. Inf. Theory 23 (1977), 337-343.

[ZIV78] Ziv, J., and Lempel, A. "Compression of individual sequences via variable-rate coding." IEEE Trans. Inf. Theory 24 (1978), 530-536.

APPENDIX A

BUCKET OCCUPANCY OF THE GRID FILE STRUCTURE

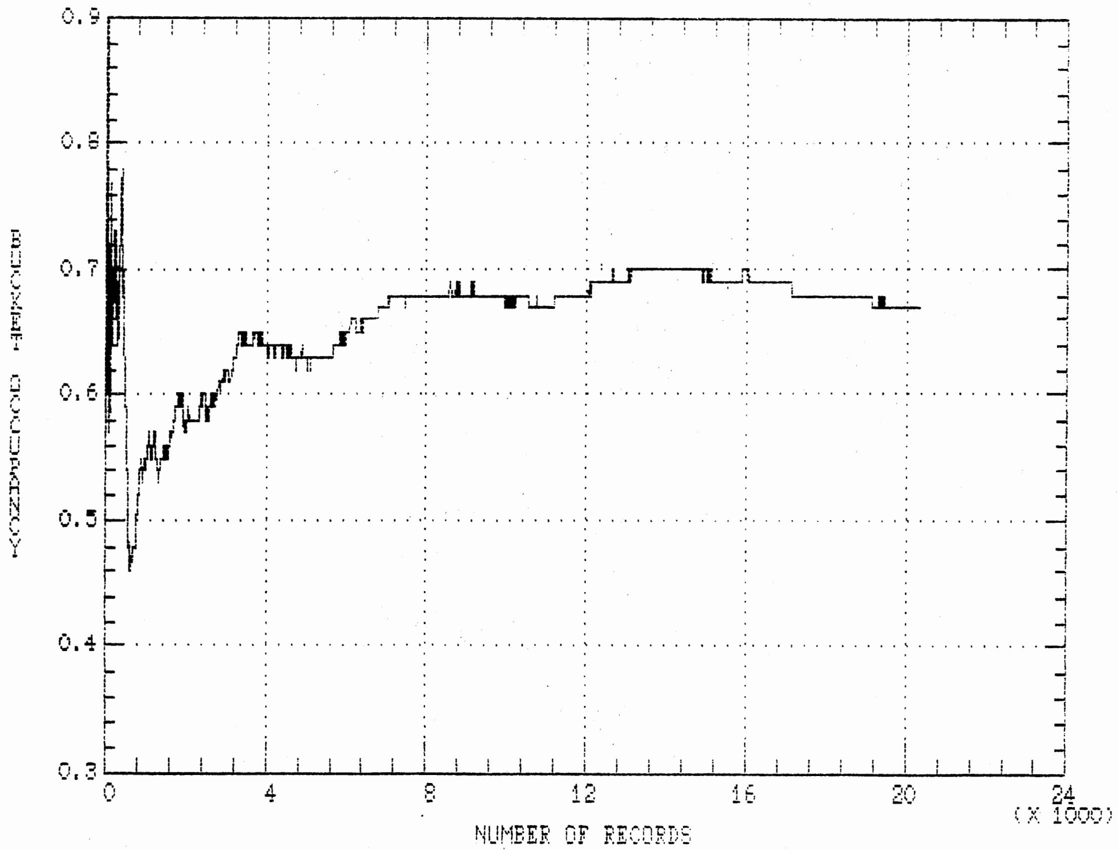


Figure A-1. Grid File Bucket Occupancy using a Randomly Generated Data File.

Figure A-1 illustrates approximately 69% occupancy at bucket size b = 20 using a randomly generated data file.

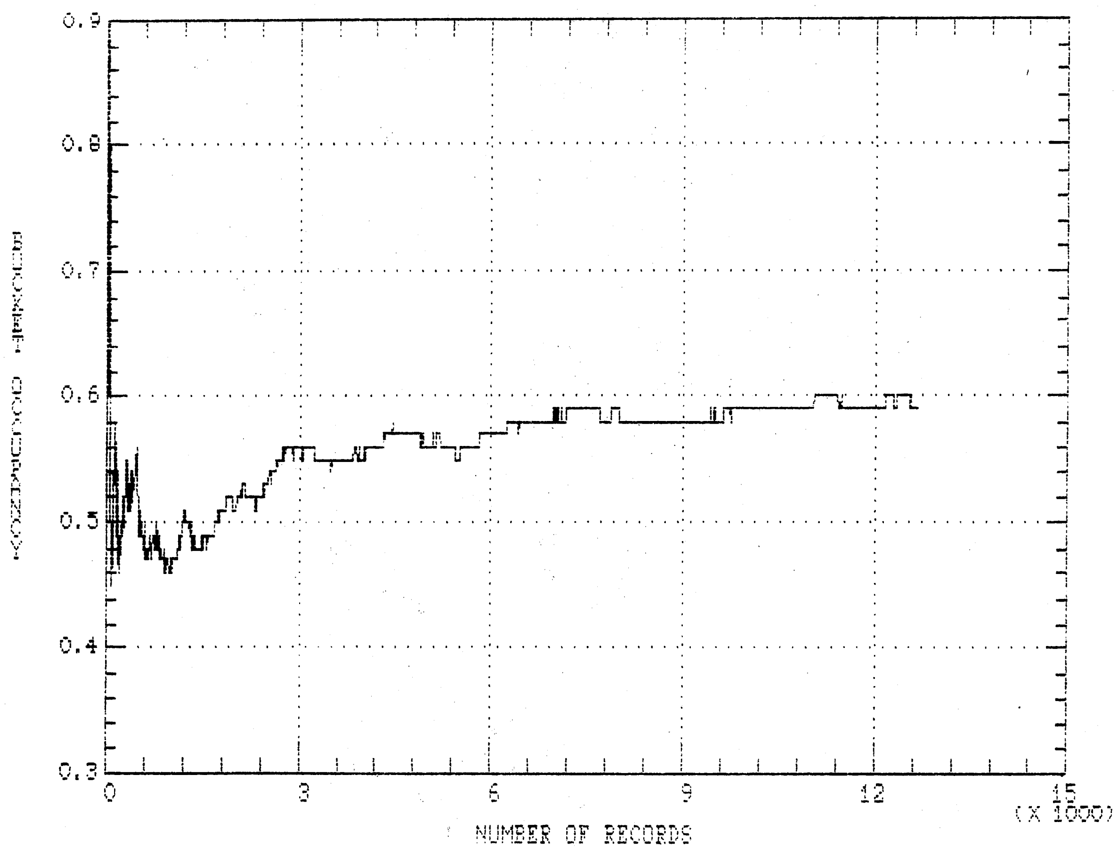


Figure A-2. Grid File Bucket Occupancy Using a Biased Data File.

Figure A-2 illustrates approximately 59% occupancy at bucket size b = 20 using a biased data file.

APPENDIX B

TOTAL FILE COMPRESSION USING THE MIDPOINT OF RECORD PARTITION METHOD

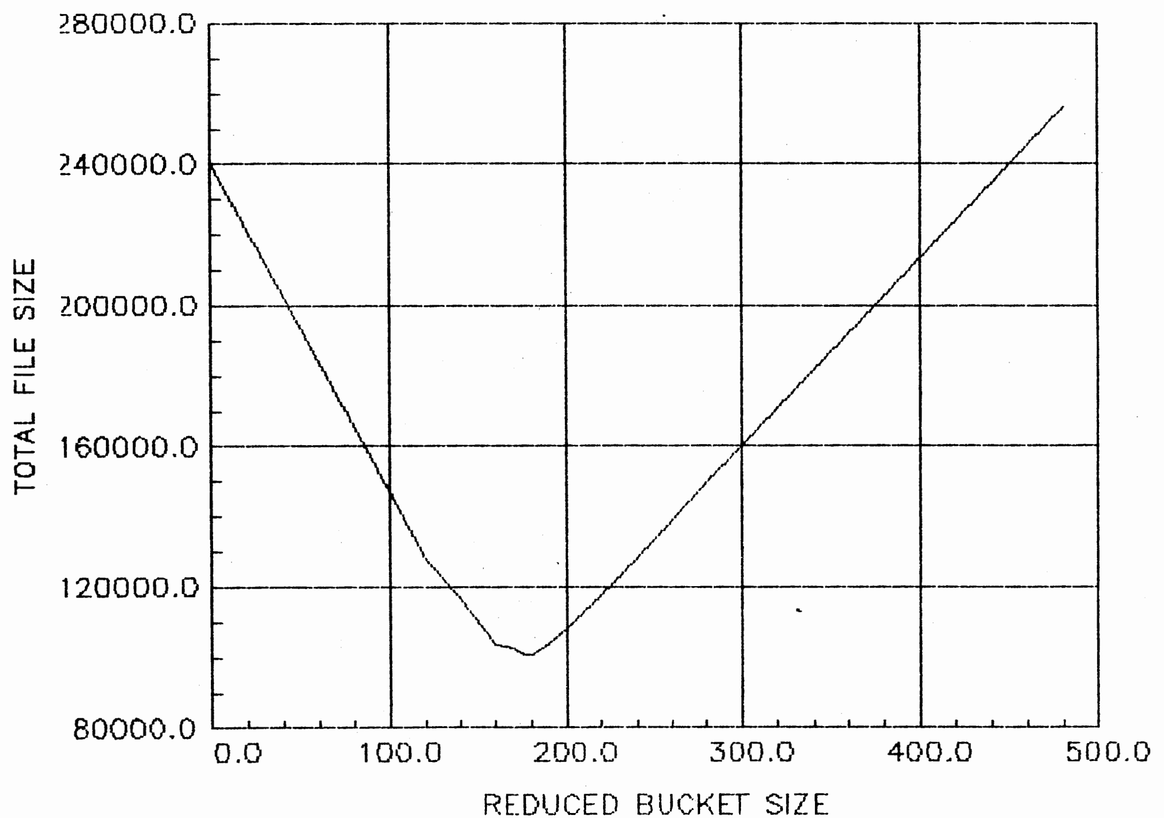


Figure B-1. Total File Size Variation using a Randomly Generated Data File.

Total file size variation at bucket size b = 10 using a randomly generated data file with midpoint of the record partition method.

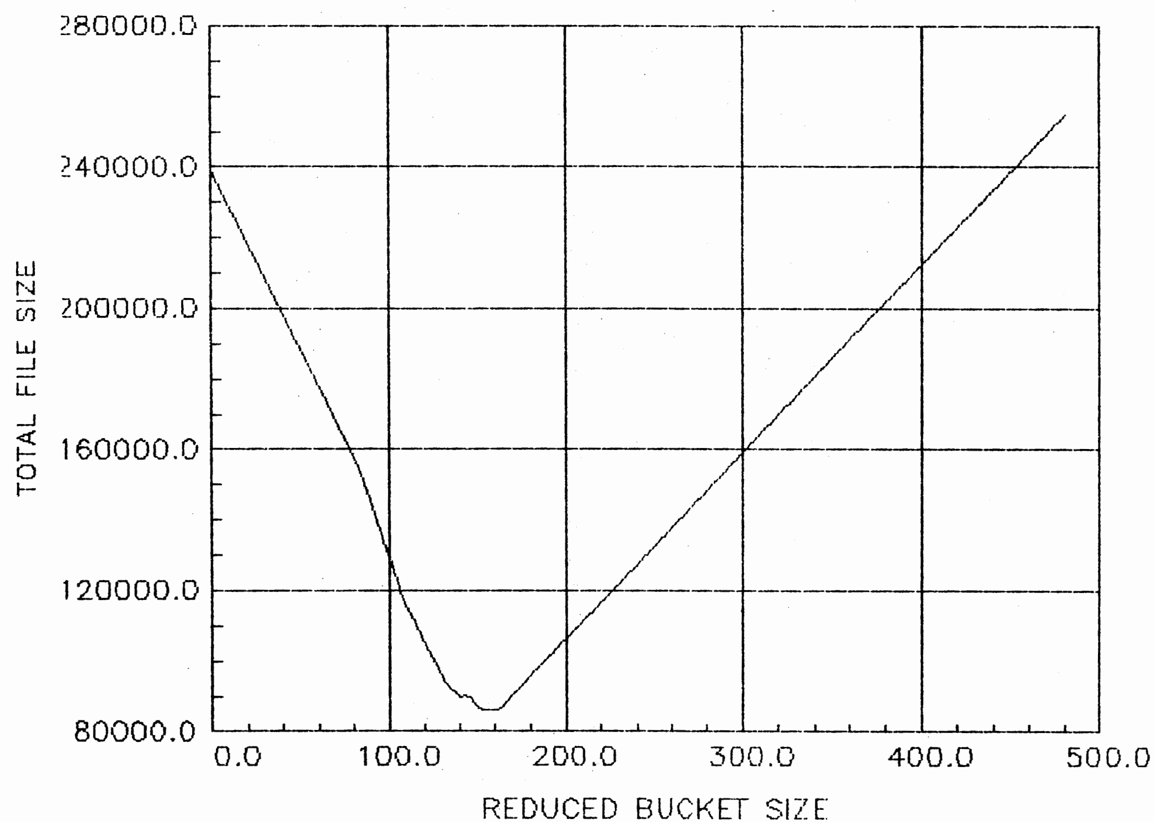


Figure B-2. Total File Size Variation using a Biased Data File.

Total file size variation at bucket size b = 10 using a biased data file with midpoint of the record partition method.

VITA

Sei Hoon Chun

Candidate for the Degree of

Master of Science

Thesis: COMPRESSION TECHNIQUES FOR THE GRID FILE

Major Field: Computing and Information Science

Biographical:

Personal Data: Born in Seoul, Korea, January, 1960, the son of Mr. Yung Il Chun and Jung Eui Chun.

Education: Graduated from Kyung-Bock High School, Seoul, Korea, in February, 1978; received Bachelor Science degree in Electrical Engineering from Hanyang University in February, 1985; completed the requirements for the Master of Science degree at Oklahoma State University in December, 1988.

Professional Experience: Programmer, Oklahoma State University, Agronomy Department, August, 1987, to September, 1988.