AN ANALYTICAL COMPARISON OF GRID FILE AND K-D-B-TREE STRUCTURES

By

HATICE NILUFER ANLAR SARITEPE

Bachelor of Science in Electrical Engineering

Bogazici University

Istanbul, Turkey

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
December, 1987

Thesis
1987D
S245a
cop. 2

AN ANALYTICAL COMPARISON OF GRID FILE

AND K-D-B-TREE STRUCTURES

Thesis Approved:

M J Folk
Thesis Adviser

D D Fisher

D. E. Hedrick

Norman N. Durham
Dean of the Graduate College

1291049

ACKNOWLEDGMENTS

I would like to express my deepest appreciation and respect to my major adviser, Dr. Michael J. Folk, for his patient guidance, concern and invaluable help in completing this thesis. I also thank Dr. Donald D. Fisher and Dr. George E. Hedrick for serving on my committee and for their encouragement during the course of my studies.

I am deeply indebted to Dr. Mete Oner for his help in the problems related to statistics and a careful reading of the thesis.

I owe my special thanks to my parents Nakip and Nuran Anlar, my sister Nilgun, for their love and moral support. Most of all I thank my husband Selcuk for his love, understanding and constant support.

TABLE OF CONTENTS

Chapter		Page
I.	INTRODUCTION	1
II.	REVIEW OF LITERATURE	3
III.	DESCRIPTION OF GRID FILE AND K-D-B-TREE STRUCTURES	8
	Introduction	8
	Grid File	9
	K-d-B-tree	11
	Performance Evaluation Criteria	12
	Cost of a Search	12
	Cost of an Insertion	14
	Memory Utilization	15
	Basic Parameters and Relationships	15
	Common Parameters and Relationships	15
	Relationships for Grid File	16
	Relationships for K-d-B-tree	17
IV.	QUERY PERFORMANCE	19
	Classification of Queries	19
	Query Performance of Grid File	22
	FP Case	23
	PP Case	24
	FR Case	24
	PR Case	24
	FF Case	25
	PF Case	25
	Query Performance of K-d-B-tree	26
	FP Case	27
	PP Case	25
	FR Case	28
	PR Case	29
	FF Case	29
	PF Case	29
	Comparisons	30
V.	INSERTION PERFORMANCE	50
	Introduction	50
	Measuring the Cost of an Insertion	51

Chapter Page

Probability of Splitting. 52
Cost of Splitting 56
Insertion Performance: Results. 60

VI. MEMORY UTILIZATION 62

Introduction. 62
Bucket Utilization. 62
Size of Access Mechanism. 63
Comparison. 66

VII. CONCLUSIONS. 73

Summary and Conclusions 73
Suggestions for Future Work 75

BIBLIOGRAPHY. 77

APPENDIXES. 79

APPENDIX A - NUMBER OF ACCESSED INTERVALS IN A RANGE QUERY . 80

APPENDIX B - BUCKET OCCUPANCY AT SPLITTING 85

APPENDIX C - INSERTIONS WITH SPLITTING 87

APPENDIX D - TABLE OF SYMBOLS. 91

LIST OF TABLES

Table   Page

I. Classification of Query Types 20

II. Comparison of Block Accesses: FP Queries (N=100000, c=100) 33

III. Comparison of Block Accesses: FR Queries (N=100000, c=100) 34

IV. Comparison of Block Accesses: FR Queries (f=1, c=100). 35

V. Summary of Block Access Formulas for Queries 48

VI. Number of Buckets 64

VII. Comparison of Access Mechanism Sizes (c=100) 69

VIII. Comparison of Access Mechanism Sizes (N=100000). 71

IX. Results of Numeric Experiments. 86

X. Table of Symbols. 91

LIST OF FIGURES

Figure | Page

1. A Convex Assignment of Grid Blocks to Buckets. 9

2. A Search for a Record by Using a Grid Directory. 10

3. Space Partitioning of a K-d-B-tree 11

4. A Search for a Record by Using a K-d-B-tree (k=2, m=3, c=4, N=26) 13

5. Comparison of Block Accesses: FR Case (Database Size 10000). . 36

6. Comparison of Block Accesses: FR Case (Database Size 30000). . 37

7. Comparison of Block Accesses: FR Case (Database Size 100000) . 38

8. Comparison of Block Accesses: FR Case (Database Size 300000) . 39

9. Comparison of Block Accesses: FR Case (Database Size 1000000). 40

10. Effect of Cycles on K-d-B-tree Bucket Accesses (Database Size 100000) 41

11. Effect of Cycles on K-d-B-tree Node Accesses (Database Size 100000) 42

12. Effect of Dimension on Grid File Node Accesses 43

13. Effect of Dimension on K-d-B-tree Node Accesses. 44

14. Comparison of Block Accesses: FR Case (Dimension 2). 45

15. Comparison of Block Accesses: FR Case (Dimension 4). 46

16. Comparison of Block Accesses: FR Case (Dimension 8). 47

17. Distribution of Bucket Occupancy Ratio at Splitting. 54

18. Bucket Occupancy Ratio at Splitting. 57

19. Number of Intervals Spanned in a Range Query (n=7) 83

20. Expected Number of Intervals in a Range Query. 84

Figure Page

21. Insertions with Splitting (Grid File). 89

22. Insertions with Splitting (K-d-B-tree) 90

CHAPTER I

INTRODUCTION

The grid file and the k-d-B-tree are two dynamic multikey access techniques developed in recent years. Separate studies have been reported on these structures from the point of view of examining their efficacy in generating, accessing, and maintaining data files. The conditions under which these two structures have been examined, as well as the methodologies used in these examinations are different for the two structures. Consequently, conclusions regarding their relative performance and suitability to specific applications can not be derived easily from the results of published studies. Therefore it is necessary to investigate the performance of the two file structures on a number of essential features using the same general criteria.

The objectives of the studies reported in this thesis are to establish performance evaluation criteria for comparing the two structures, and to apply these criteria to both structures to obtain specific information on their efficiency and other relevant characteristics.

An analytical approach has been followed in this study in order to derive general formulas that can be used to estimate the relative performance of the two structures. Some numerical results have also been presented to illustrate the usage of these formulas as well as to obtain some indications for relative performance. At the beginning of

this research we considered comparing the results of simulations or simplified impementations based on a hypothetical database. It is believed that the analytical approach is more suitable for the purposes of this study due to a number of reasons: programming details could affect the comparisons, many runs are needed for a statistically meaningful result in a simulation, and the properties of the hypothetical database may not be typical for a real application.

Chapter II is a general review of literature on multikey access techniques. Chapter III contains a brief description of the grid file and k-d-B-tree structures. A set of criteria for comparing their performance is developed in this chapter. The chapter concludes by giving the parameters and relationships for both structures.

Chapters IV, V, and VI contain the analyses and discussions of the three basic aspects of performance: query efficiency, insertion performance, and memory utilization. Chapter VII summarizes the findings of these studies.

CHAPTER II

REVIEW OF LITERATURE

In recent years, the increasing usage of databases and integrated information systems has encouraged the development of file structures specifically suited to accessing records by combinations of attribute values. The method of using several attributes for accessing records is called multikey access, and records specified with several keys are called multidimensional data. The early development of file structures that provide multikey access to records are extensions of file structures originally designed for single-key access. Most balanced structures for single-key data rely on a total ordering of the set of key values. Since natural total orders of multidimensional data do not exist, the design of balanced data structures for multidimensional data is significantly more difficult.

Inverted files were among the earliest of file structures designed for multikey access [1]. Since they have been used in most applications they have been accepted as a standard to evaluate alternative approaches. Inverted files are well suited for accessing records on the basis of Boolean conditions on the attributes, but they exhibit some drawbacks. First, retrieval of the inverted lists may require an excessive number of disk accesses. Second, the overhead required for insertions and deletions can become prohibitive in terms of space and time. Finally, in environments where several keys are equally

significant, a file structure that treats all significant keys symmetrically is appealing.

In the remainder of this section we briefly describe a variety of multikey file structures, each designed to perform better than an inverted file in at least some circumstances. Many of the approaches are generalizations of well-known single-key file structures.

Several generalizations of inverted files have been proposed. Lum [2] describes "combined indices", in which several attributes are concatenated in various orders and then treated as a single, aggregate key. If more than three attributes are combined, both the storage space and update time become excessive. By combining them in groups of three, however, the number of disk accesses to retrieve inverted lists can be reduced substantially, at the cost of some increased complexity [3]. Vallarino [4] describes another generalization of inverted lists called "compressed bitmaps". Bit-encoded inverted lists are the basis of this structure. They form a large sparse bit array, which is then represented in highly compressed form and used to locate records specified by a selection condition. Another organization that exploits compression in providing multikey access is the "transposed file" organization [51]. In this organization, vectors consisting of the values of a particular attribute for all records are stored in a highly compressed form. Thus, the retrievals and updates that refer to only a few attributes do not involve memory transfers of irrelevant attributes. This approach is most effective when the majority of operations deal with a significant portion of the records (i.e., one to three percent) and selection conditions involve only a few attributes.

Rothnie and Lozano [6] describe a generalization of hashing in which a bucket address for a record is formed by concatenating the results of hash functions, each of which is applied to the value of one key. A critical design decision in setting up such a "multikey hash file" is the determination of the number of the bits to be allocated to represent the hashed value of each attribute. The more attribute values specified, the smaller the number of buckets that need to be accessed in order to obtain the required records [7]. Because it is difficult to specify a combination of hash functions that lead to a uniform occupancy of buckets, it is necessary to tolerate either a low bucket occupancy, or a high likelihood that buckets overflow (more than one storage block is needed to hold the records corresponding to a single bucket). Also, like most hashing schemes, multikey hashing is inappropriate when the selection condition involves ranges of values rather than specific values.

Various generalizations of tree structured indices permit multikey access to files. Quad trees are a two-attribute generalization of binary search trees [8]. The straightforward generalization to k dimensions is impractical because the tree nodes become large and contain many nil pointers. These problems are avoided in k-d-trees [9, 10], which can be thought of as k-dimensional generalization of binary search trees. Each level of the tree is associated with a different key in turn. K-d-trees are efficient for large and very large databases.

Similarly, binary TRIEs can be generalized to support multikey access [1]. This is achieved by representing each attribute value as a bit string and interleaving these strings. The result is then used as the key in a standard binary TRIE.

The "multiple-attribute tree" database organization orders the records lexicographically on the key fields, with the more significant attributes placed toward the higher end of the sorting field [11]. Then the key fields are separated from the records and organized into a doubly-chained tree. The tree can then be used to locate all relevant records for a given query. If both the number of records and the number of attributes are large, several disk accesses may be required to locate records satisfying specified constraints on key values.

Casey describes a complex tree-based multikey access structure in which records are grouped according to the frequency in which they are retrieved together [12]. "Superimposed coding" is used in each node to characterize the records below the node in the tree. Probably because of its complexity, this organization has not been widely used in practice. The importance of this structure is due to the fact that, more than with any other multikey file structure, the selection conditions used in accessing the file influence the its organization.

A "Quintary tree" is a file structure intended to provide faster access then other tree-based multikey file structures, at the cost of requiring more space [13]. Quintary trees consist of k levels, corresponding to the k attributes in decreasing order of importance. Each level resembles a binary tree branching on the values of the corresponding attribute.

Robinson [14] describes "k-d-B-trees" which combine properties of both B-trees and k-d-trees. It is a balanced multiway tree and each level of the tree corresponds to a different attribute. Internal nodes reflect the partitioning of the search space into nonoverlapping regions. Performance of k-d-B-trees on partially specified queries is

explained in [15]. Similar to k-d-B-trees, "multidimensional B-trees" and other related approaches are explained in [16, 17].

Along with k-d-B-tree, other multikey organizations have been proposed recently that are also based on the idea of partitioning k-dimensional space and then storing the records corresponding to each cell of the partition in a single block of secondary storage. One such organization is the "multidimensional directory" suggested by Liou and Yao [18]. Attributes are ordered by priority, and higher priorities are associated with the attributes that appear more often in the queries. A multidimensional directory, which contains one entry per secondary storage block, is used for retrieval of records.

The "grid file" is also based on the idea of partitioning the search space by treating all dimensions symmetrically [19]. A dynamic grid directory is utilized to locate the records on the secondary storage blocks. This file system adapts gracefully to its contents under insertions and deletions, and thus achieves an upper bound of two disk accesses for single record retrieval; it also handles range queries efficiently.

Multipaging, dynamic multipaging and interpolation based index maintenance are some other recent multikey access schemes mentioned in [19] that utilize grid partitions of search space in ways similar to grid files [20, 21, 22].

CHAPTER III

DESCRIPTION OF GRID FILE AND K-D-B-TREE STRUCTURES

Introduction

Searching techniques for multikey access can usually be divided into the following two categories:

a. techniques that organize the specific set of data to be stored and,

b. techniques that organize the search space to which the data belongs.

Comparative search techniques, such as different tree structures fall into the first category. In these structures, the boundaries between different regions of the search space are determined by values of data that are to be stored. On the other hand, address computation techniques, such as hash files fall into the second category. K-d-B-tree and grid file are two good examples of these two categories respectively. Both structures partition the search space into subspaces, down to the record level on the secondary storage. But, the way they do this partitioning is different and much can be learned by comparing these two structures.

Grid File

The grid file is based on the use of "rectangular" partitions that divide the search space into regions. Each region has the shape of a

rectangle in a two-dimensional space and the shape of a box in a three-dimensional space. In a k-dimensional space these regions may be visualized as k-dimensional rectangles. Each region boundary cuts the entire search space into two. The grid file assumes that the attributes are independent so that the partitions are fully utilized. Partitioning of the record space is done by imposing a number of intervals on each dimension. The intersection of these intervals divides the record space into blocks, called "grid blocks." All records in one grid block are stored in the same bucket, but it is possible for several grid blocks to share the same bucket, as long as the union of these grid blocks forms a region in the record space. The regions of buckets are pairwise disjoint, together they span the entire space of records (Figure 1).

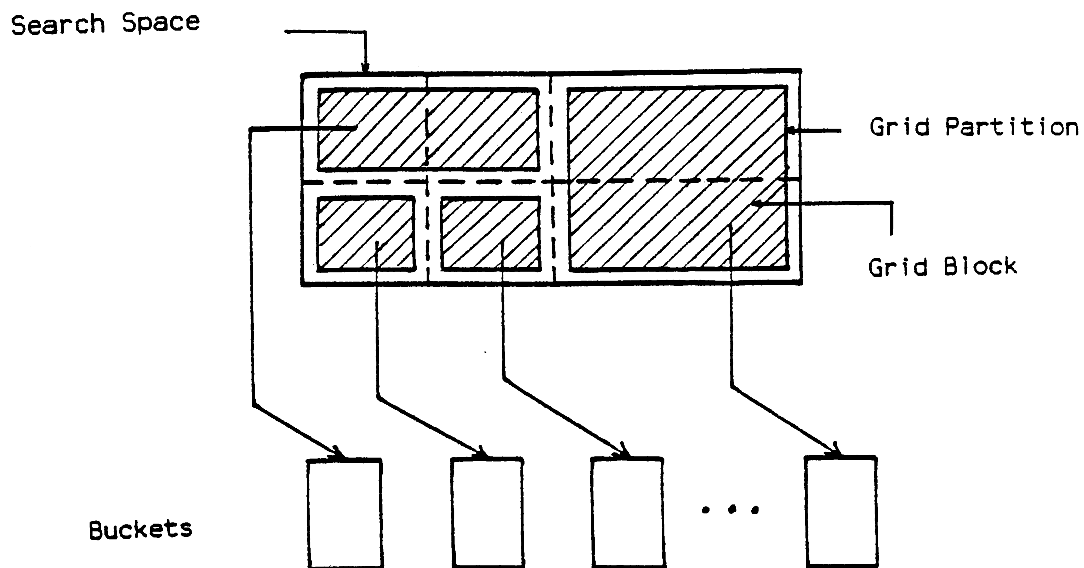


Figure 1. Assignment of Grid Blocks to Buckets

The dynamic correspondence between the grid blocks in the record space and data buckets is provided by a "grid directory." The grid directory consists of two parts:

a. k one-dimensional arrays called linear scales, and

b. one k-dimensional array called the grid array.

Each linear scale defines the partitioning of each dimension and is divided into a certain number of intervals. Linear scales are used as indexes to the grid array. Elements of the grid array are the pointers to the data buckets and are in one-to-one correspondence with the grid blocks of the partition. To access a record, first the linear scales are searched to find the related intervals for the key values. These intervals are used to locate the grid block in the grid directory. That grid block contains the address of the bucket where the record is stored. An example of a search for a record by using a grid directory is shown in Figure 2.

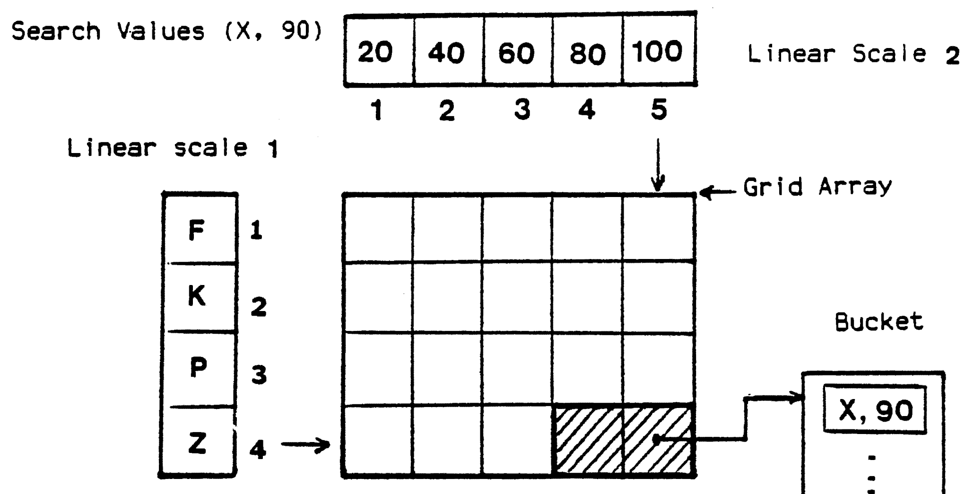


Figure 2. A Search for a Record by Using a Grid directory.

The grid file is designed to handle efficiently a collection of records with a modest number of search attributes ($k<10$) whose domains are large and linearly ordered [19]. If we define a bucket as a storage unit that contains records, research indicates that grid file gives the best performance when the bucket capacity (c) is between 10 and 1000 records [19].

K-d-B-tree

A k-d-B-tree is a balanced multiway search tree with fixed sized nodes. K-d-B-trees partition the search space in a manner similar to k-d-trees: the search space is divided into subspaces based on a comparison with some element of a single domain (Figure 3).

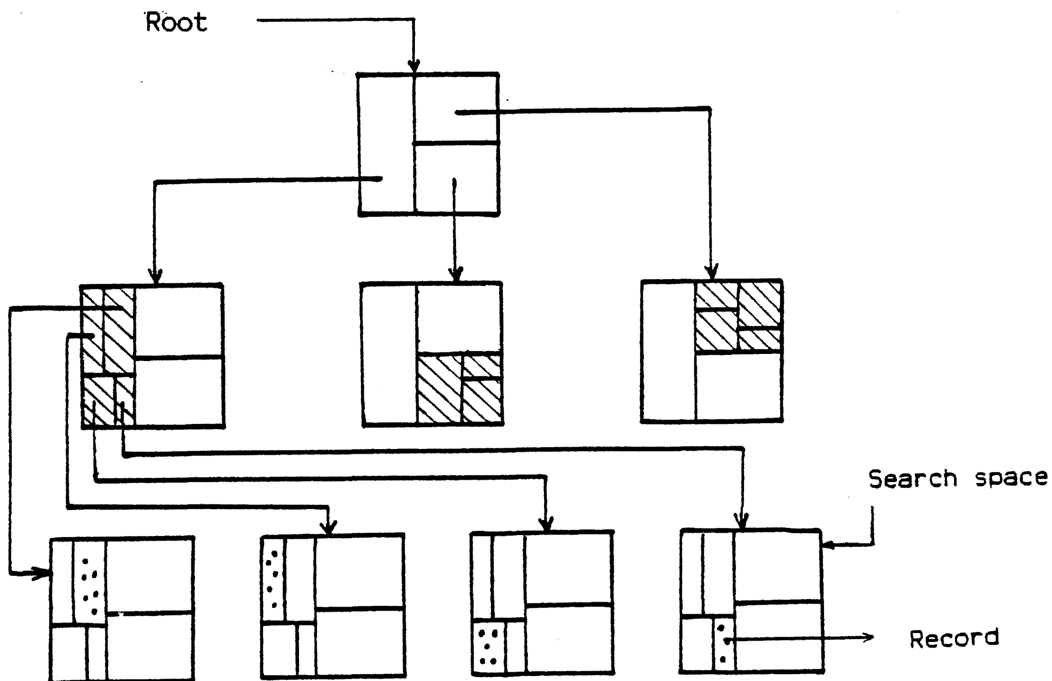


Figure 3. Space Partitioning of a K-d-B-tree

A k-d-B-tree has the following properties [14]. Different levels of the tree correspond to different keys. The root of the tree represents the partitioning of the entire k-dimensional search space with respect to the first key. As in a B-tree, a k-d-B-tree consists of a root and a collection of nodes. Each node in a k-d-B-tree, including the root node, contains key values that define the regions on which the next level of the tree is based, and the pointers for each region that point to the nodes of the next level. The leaf nodes have the same structure, but instead of node-pointers, there are bucket addresses. The path length from the root node to a leaf node is the same for all leaves. The regions defined in every node are disjoint and their union is also a region.

An example of a search for a record by using a k-d-B-tree is shown in Figure 4.

Performance Evaluation Criteria

In this section we discuss possible criteria for comparing the relative performance of the grid file and k-d-B-tree in queries, insertions, and memory utilization.

Cost of a Search

To compare the cost of a search in a grid file accessed database and in a k-d-B-tree accessed database, a common measure has to be defined. The basic unit of measure used in this study is the number of block accesses required to respond to a given query. The types of operations that involve block accesses for a grid file are retrieval of linear scales, retrieval of the grid array, and retrieval of buckets.

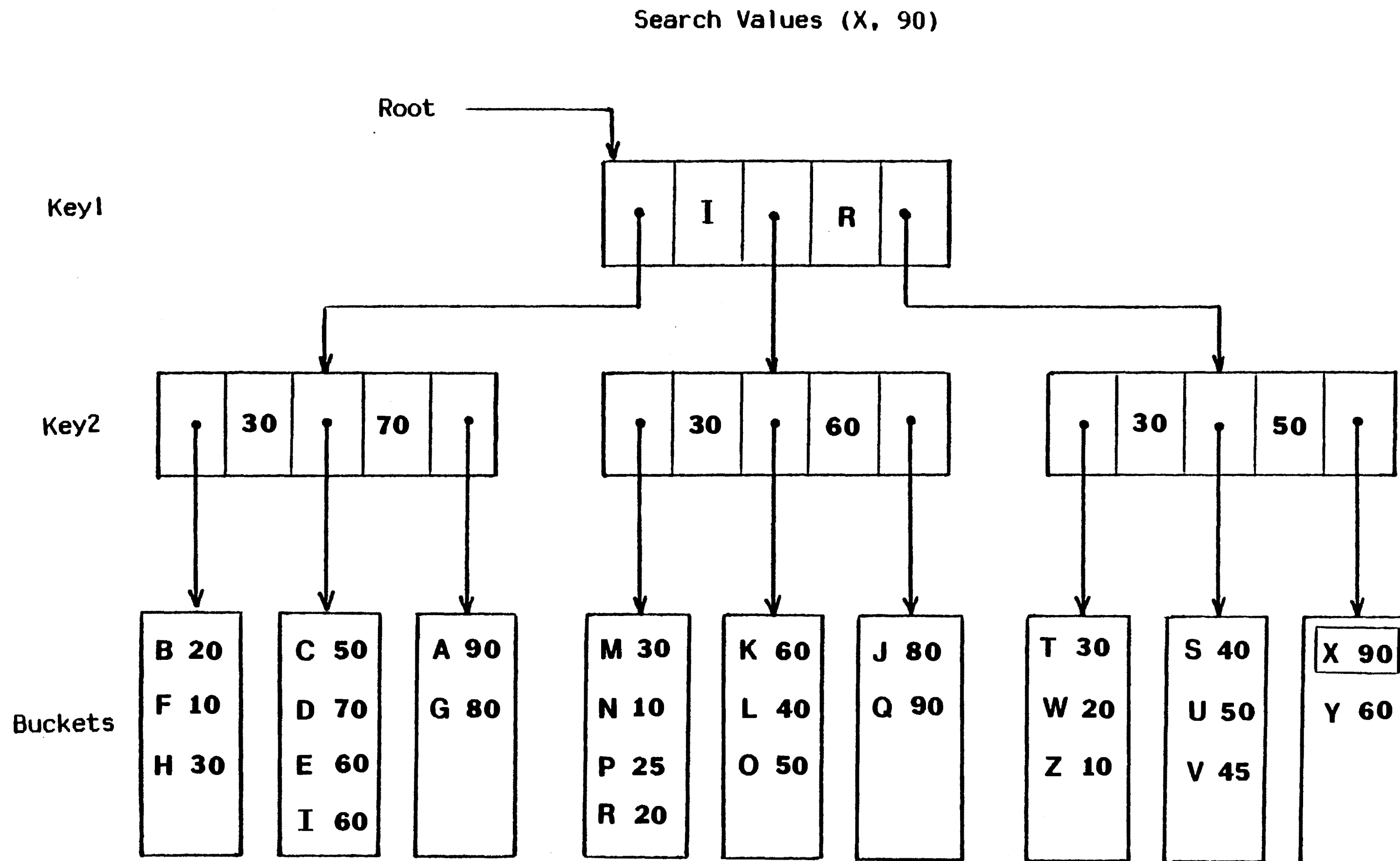


Figure 4. A Search for a Record by Using a K-d-B-tree (k=2, m=3, c=4, N=26)

For a k-d-B-tree the type of operations that involve block accesses are retrieval of nodes and retrieval of buckets. There are different types of queries. Each type of query search for a grid file can be compared to the same type of query search using a k-d-B-tree.

Once the expected counts of the block accesses are found, then the cost of each type of search can be computed relative to the costs of the other types of searches.

Cost of an Insertion

When a new record is inserted into the database it is necessary to determine whether the bucket in which it belongs has available space. If the bucket has enough room then the new record is inserted without any complication. But if the bucket is already full, it causes overflow and splitting becomes necessary. Even though splitting is assumed to occur rarely, when measuring cost it needs to be considered.

To calculate the average cost of an insertion, we may utilize the following probability formula:

$$E(\text{insertion}) = C1\ P(x) + C2\ (1 - P(x)) \tag{3.1}$$

where

E(insertion) is the expected cost of an insertion,

C1 is the cost of an insertion with splitting,

C2 is the cost of an insertion without splitting.

P(x) represents the probability of an occurrence of a splitting case, and 1 - P(x) the probability that splitting will not occur.

Since splitting is rare we may expect to have a small probability value for P(x). C1 and C2 can be considered as the maximum and the minimum costs for the insertion respectively.

Memory Utilization

Memory utilization can be studied in two parts: bucket occupancy and the memory required by the grid file and the k-d-B-tree structures themselves.

The bucket occupancy ratio is a good measure of the memory utilization efficiency of an access mechanism. Alternatively, one may consider the number of buckets required to hold the records of a given database.

In a grid file the amount of memory required for the structure itself may be calculated by considering the sizes of the linear scales and the grid array. In the k-d-B-tree case this involves the amount of memory required to hold the information that is contained in nodes; pointers and key values.

Basic Parameters and Relationships

We will first define a consistent set of parameters governing the basic features of the grid file and k-d-B-tree, and then study their inter-relationships. These relationships will be needed in comparison studies that will follow.

In the formulations it will be assumed that the key fields (attributes) are not correlated.

Common Parameters and Relationships

Parameters common to both the grid file and k-d-B-tree are

N = total number of records in the database,

b = number of buckets,

c = bucket capacity (in terms of records),

e = average bucket occupancy ratio,

k = number of keys (dimension).

The total number of records can be calculated in the same way for both structures, namely

$$N = b\ c\ e \tag{3.2}$$

Relationships for Grid File

In the grid file structure there is a linear scale associated with the domain of each key. The number of the linear scales is the same as the number of keys, that is k. If there are n_i intervals in the linear scale for key number i, then the total number of grid blocks in grid array G may be expressed as:

$$G = n_1\ n_2\ n_3\ \ldots\ n_k \tag{3.3}$$

or

$$G = \prod_{i=1}^{k} n_i \tag{3.4}$$

To obtain a working equation for an approximate analysis we may consider the special case where all linear scales have been divided into the same number of intervals, n. This simplifies Eq. (1.3) as:

$$G = n^k \tag{3.5}$$

In grid file storage organization it is common to assign more than one grid block to each bucket. If r denotes the number of grid blocks per bucket, then, by definition,

$$r = G\ /\ b \tag{3.6}$$

or, on substitution of Eq. (3.5),

$$r = n^k / b \tag{3.7}$$

A typical average value for r seems to be 2 in [19].

Solving b from Eq. (3.7) and substituting in Eq. (3.2) yields:

$$N = n^k c e_g / r \tag{3.8}$$

In the formula above, the e_g is the bucket occupancy ratio for the grid file.

Relationships for K-d-B-tree

In a k-d-B-tree it is not necessary that all nodes be of the same order. Node order can vary from one level to another, and even within a level. In this study node order, m, is assumed to be the same for the internal nodes throughout the whole tree to be able to generalize the relationships. The height of the tree is represented by h. Height of the tree is the path length from the root to the leaf level. Leaf nodes contain bucket pointers. To calculate the maximum number of buckets required we need to know the number of pointers at the leaf level. The following formula gives the maximum number of buckets in terms of the tree height h and order m.

$$b = m^h \tag{3.9}$$

To distinguish it from the grid file storage efficiency, the efficiency for the k-d-B-tree is denoted by e_k. The number of records can be expressed in the following formula:

$$N = m^h c e_k \tag{3.10}$$

The formula above includes almost all important parameters. The dimension, k, does not appear in the formula explicitly. Actually the dimension is a major factor affecting tree height, h. From the

definition of the k-d-B-tree, we can easily say that the minimum value for the height is the number of the keys used, which is k. We can not know the usage frequency of each key, but if we assume the frequency is the same for all of the keys and denote it as f, then we get the following relationship:

$$h = f\ k \quad (3.11)$$

When the frequency is the same for all the keys, then it has to be an integer. If all the keys are used only once in the partitioning of the tree then f is 1. At this point we define the concept of "cycle" which is very similar to the usage frequency. If the dimension of the tree is two, then key1 and key2 are used in sequence in the partitioning of the tree. If the keys are used more than once then the same sequence of the keys (key1, key2) continues and each sequence is called a cycle. Since frequency is assumed to be the same for all the keys, then frequency is equal to the number of cycles in the tree. So, those two terms can be used interchangeably. In this study the range value for frequency and cycle is assumed to be between 1 and 4.

CHAPTER IV

QUERY PERFORMANCE

Classification of Queries

A file is a collection of records and a record can be considered to be a collection of key values and any additional information about the item in the record. Multikey access allows the records in a file to be referenced by using any possible subset of the key fields.

The databases suited to a multikey access structure are grouped in two categories. The first group includes records whose keys (attributes) are many but their domains are small. This group is not very typical. The second and more typical group includes records characterized by a small number of keys (less than 10), but the domain of each key is large and linearly ordered. The second case will be considered in this study.

For the second case we can specify ranges by expressions of the form $l_i <= key_i <= u_i$ where l_i and u_i are the lower and upper bounds of the domain, respectively, for the key key_i. When l_i becomes the smallest and u_i becomes the largest values of the domain i, then the range becomes a "full range" covering the entire domain. Similarly, if l_i becomes equal to u_i, then the range becomes a "point".

Queries can be classified into three groups by the range of the key values according to the above definitions.

a. Point query: the requested value of key field is a single value in the domain of that key field.

b. Range query: a valid range inside the domain of the key specified for that requested key.

c. Full range query: values spanning the whole domain of that key is requested. This can be considered as a "don't care" case.

In a multidimensional search space, a record consists of many keys. For retrieval of one or more records, either all of the keys or a subset of them are specified. If the total number of keys is k, denote by k_s the number of specified keys. Therefore another classification of queries can be done according to the number of keys used in a query.

a. Partially specified query: some keys (k_s<k) are specified.

b. Fully specified query: all keys (k) are specified.

As a result of these two independent classifications, there are, in general, six combinations as shown in Table I.

TABLE I

CLASSIFICATION OF QUERY TYPES

	Point	Range	Full Range
Fully Specified	FP	FR	FF
Partially Specified	PP	PR	PF

These query types have been labeled by two-letter symbolic names for use in the discussions that follow. The first letter indicates

whether the query is Partially (P) or Fully (F) specified. The second letter indicates the range covered; P for Point, R for general range, and F for full range.

To see how these different types of queries work, a small sample database can be used. This database has information about some high school students. Key fields are last name, year of birth and GPA. Records may have more information beside key fields. Let us assume the following records are stored in this database. An example of each query type is given below.

Rec No.	Last Name	Year of Birth	GPA	Other Info.
1	Anderson	1969	2.45	...
2	Jones	1970	3.32	...
3	Marble	1971	3.87	...
4	Smith	1964	2.85	...
5	Taylor	1968	3.20	...
6	Wilson	1972	3.75	...
7	Watson	1973	2.48	...

FP Type Query

Query: Find the record for the student whose last name is Smith, born in 1964 with GPA 2.85.

Result: Record no. 4.

PP Type Query

Query: Find the information about the student whose last name is Wilson and born in 1972.

Result: Record no. 6.

FR Type Query

Query: Find the student records with last names between M.. and T.., born between 1965 and 1975, and have GPAs between 2.50 and 3.50.

Result: Records 3,4 and 5.

PR Type Query

Query: Find the student records with last names starting with W and born between 1970 and 1975.

Result: Records 6 and 7.

FF Type Query

Query: Find the student records with last names between A.. and Z... and born between 1960 and 1980 and with GPA between 1.00 and 4.00.

Result: Records 1, 2, 3, 4, 5, 6, and 7.

PF Type Query

Query: Find the records for the students born between 1960 and 1980.

Result: Records 1, 2, 3, 4, 5, 6, and 7.

Query Performance of Grid File

In the grid file structure, the retrieval of records involves three different types of block accesses for all types of the queries: linear scales, grid blocks, buckets.

According to the nature of each query type the number of block accesses involved in each step is expected to be different for the cases above. To calculate the cost of each query, the cost associated with each type of block access needs to be considered. The following notation will be used to indicate the different costs involved in the grid file structure:

C_l = cost of a linear scale access,

C_g = cost of a grid block access,

C_b = cost of a bucket access.

Then the cost of a query can be computed as

$$E_g(\text{query}) = C_l\ a_l + C_g\ a_g + C_b\ a_b \qquad (4.1)$$

where a_l, a_g, and a_b denote the number of linear scales, grid blocks, and buckets accessed, respectively.

To obtain the expected cost of each query type, each case needs to be examined separately to obtain the probable number of accesses.

In range type queries (FR and PR), it is necessary to estimate the number of intervals that will be covered by a "typical" range query specification. If a linear scale has been divided into n intervals, then the number of covered intervals will range from 1 through n. The expected value of these covered intervals is called n_a in the following. Naturally n_a is related to n. This relationship is derived probabilistically and explained in Appendix A. It has also been evaluated numerically, and the following curve fit has been derived for use in the block access count formulas,

$$n_a = 0.80\ n^{0.71}, \quad n < 9 \qquad (4.2)$$

$$n_a = 0.57\ n^{0.87}, \quad n >= 9 \qquad (4.3)$$

FP Case

Since all linear scales are searched in this type of query, the number of linear scales accessed is k. FP type of query specifies point values for all keys which defines at most one unique record in the data base. Thus in this case only one grid block and one bucket need to be accessed:

$$a_l = k \qquad (4.4)$$

$$a_g = 1 \qquad (4.5)$$

$$a_b = 1 \qquad (4.6)$$

PP Case

In this case k_s keys are specified, therefore the same number of linear scales will be accessed:

$$a_l = k_s \qquad (4.7)$$

The number of grid blocks to be accessed in this case is determined by the "don't care" keys, because there will be one linear scale interval for each of the specified keys while the entire domain of a "don't care" key has to be searched. Thus,

$$a_g = n^{(k-k_s)} \qquad (4.8)$$

The number of buckets to be accessed is equal to the number of grid blocks divided by r, the number of grid blocks per bucket:

$$a_b = n^{(k-k_s)} / r \qquad (4.9)$$

FR Case

Since all keys are specified in this case, all linear scales are searched:

$$a_l = k \qquad (4.10)$$

For range queries it is assumed that a range covers n_a intervals on each linear scale. So the number of grid blocks to be accessed is:

$$a_g = n_a^k \qquad (4.11)$$

The number of buckets to be accessed is equal to the number of grid blocks divided by the number of grid blocks per bucket, r:

$$a_b = n_a^k / r \qquad (4.12)$$

PR Case

In this case k_s keys are specified, therefore the same number of linear scales will be accessed:

$$a_l = k_s \qquad (4.13)$$

For specified keys only n_a intervals (out of n intervals) in the related linear scale will be accessed. But for the unspecified $(k-k_s)$ keys, all the intervals in the corresponding linear scale will have to be considered. So the number of grid blocks to be accessed is determined by:

$$a_g = n_a^{k_s} \; n^{(k-k_s)} \qquad (4.14)$$

The number of buckets to be accessed is equal to the number of grid blocks divided by r:

$$a_b = n_a^{k_s} \; n^{(k-k_s)} \; / \; r \qquad (4.15)$$

<u>FF Case</u>

In this case all keys are used and their entire domains are covered. This simply means that the entire database will be retrieved. So the number of linear scales will be k:

$$a_l = k \qquad (4.16)$$

Also number of grid blocks and number of buckets will be equal to their maximum numbers:

$$a_g = n^k \qquad (4.17)$$

$$a_b = n^k \; / \; r \qquad (4.18)$$

<u>PF Case</u>

In this case some keys are specified. So only these linear scales are accessed:

$$a_l = k_s \qquad (4.19)$$

Specified keys cover their full domains. Since unspecified keys are considered "don't care" keys they also cover their full domains. Then

the situation becomes the same as in FF case. Thus,

$$a_g = n^k \tag{4.20}$$

$$a_b = n^k / r \tag{4.21}$$

Query Performance of k-d-B-tree

In the k-d-B-Tree, the retrieval of records involves accesses of the nodes and the buckets.

In order to compare the k-d-B-tree with the grid file it will be assumed that: (1) a node in k-d-B-tree corresponds roughly to one linear scale or one grid block of a grid file, (2) buckets have the same size in both, and (3) the cost of accessing one bucket is the same in both cases. Here, the number of node and bucket accesses involved for each of the cases above shall be calculated. These counts will be used in the numerical comparisons.

To calculate the cost of a query, the costs associated with a node and a bucket access need to be considered. The following notation will be used to indicate the different costs involved in the k-d-B-tree structure:

C_n = cost of a node access,

C_b = cost of a bucket access.

The expected cost of a query (of any type) can then be expressed as

$$E_k(\text{query}) = C_n\, a_n + C_b\, a_b \tag{4.22}$$

To obtain the expected cost of each query type, the number of node and bucket accesses are needed. This is done in the following for each query type. The root node is always accessed in each query, therefore, it will be included in the counts for the number of node accesses. In

the following, the "1" at the beginning of each node count expression corresponds to the root node.

In range type queries (FR and PR), we need to estimate the number of pointers that will be accessed by a "typical" query specification. If there are m pointers in a node, then any given query may require the use of 1, 2, ..., or m pointers at that node. The expected value of this is called m_a in the following. The relationship between m and m_a is the same as that between n and n_a that has been derived earlier for the grid file (see Appendix A). Thus,

$$m_a = 0.80\ m^{0.71}, \quad m < 9 \tag{4.23}$$

$$m_a = 0.57\ m^{0.87}, \quad m >= 9 \tag{4.24}$$

FP Case

In a fully specified point query there is only one unique record searched. Therefore only one bucket will be accessed:

$$a_b = 1 \tag{4.25}$$

Also, there will be a single path to be followed, starting from the root node down to the bucket that contains the record; thus, the number of nodes that will be encountered in this search will be equal to the height of the tree, h, which is equal to fk. In the following formula the first term stands for the root node and second term represents the other nodes.

$$a_n = 1 + (fk-1) = fk \tag{4.26}$$

PP Case

In this case some keys are specified by their point values. The number of nodes to be accessed is determined by the "don't care" keys.

For those keys all pointers in the related nodes are used to go to the next level in the tree. In the following formula the first term stands for the root node, the second term is the sum of the nodes accessed in the first cycle and third term is the number of the nodes accessed from the second to f^{th} cycle in the tree.

$$a_n = 1 + \sum_{i=1}^{k-1} m^{i(1-k_s/k)} + \sum_{j=1}^{f-1} \sum_{i=1}^{k} m^{(jk-i)} \qquad (4.27)$$

In order to calculate the number of buckets to be accessed, only the "don't care" keys have to be considered in the first cycle. In the other cycles all keys are considered:

$$a_b = m^{(k-k_s)} \quad m^{(f-1)k} \qquad (4.28)$$

FR Case

All keys are involved in this case. It is assumed that a range query covers m_a pointers of each node. This situation is only valid for the first cycle since it defines the initial partitioning of the search space. Other cycles refine the partitions of the first cycle therefore all pointers in the nodes will be accessed for those cycles. In the following formula the root node, nodes of first cycle and the nodes of other cycles are represented by the first, second and last terms respectively.

$$a_n = 1 + \sum_{i=1}^{k-1} m_a{}^{i} + m_a{}^{k} \sum_{j=1}^{f-1} \sum_{i=1}^{k} m^{(jk-i)} \qquad (4.29)$$

The number of buckets to be accessed will be:

$$a_b = m_a{}^{k} \; m^{(f-1)k} \qquad (4.30)$$

PR Case

Similar to FR case, the number of pointers accessed in a node is m_a for the specified keys and m for the "don't care" keys. Root node, nodes of first cycle and nodes of other cycles are represented by the first, second and last terms of the following formula respectively.

$$a_n = 1 + \sum_{i=1}^{k-1} m_a{}^{i(k_s/k)} \; m^{i(1-k_s/k)} + m_a{}^k \sum_{j=1}^{f-1} \sum_{i=1}^{k} m^{(jk-i)} \qquad (4.31)$$

The number of buckets to be accessed is:

$$a_b = m_a{}^{k_s} \; m^{(k-k_s)} \; m^{(f-1)k} \qquad (4.32)$$

FF Case

Since all keys are specified and their full ranges are covered, this case means that the entire database will be retrieved. The number of nodes to be accessed is equal to the total number of the nodes in the whole tree:

$$a_n = 1 + \sum_{i=1}^{fk-1} m^i = (m^{fk}-1) \; / \; (m-1) \qquad (4.33)$$

Similarly the number of buckets to be accessed is equal to the maximum number of buckets that the structure can use:

$$a_b = m^{(fk)} \qquad (4.34)$$

PF Case

For specified keys, since the full range is requested, all the pointers of the related nodes are used. For unspecified keys, full ranges are covered by definition. Consequently, this case becomes the

same as PP case where the number of nodes and number of buckets are given by:

$$a_n = 1 + \sum_{i=1}^{fk-1} m^i = (m^{fk}-1) / (m-1) \qquad (4.35)$$

$$a_b = m^{(fk)} \qquad (4.36)$$

Comparisons

In order to investigate the comparative query performance of the grid file and the k-d-B-tree a number of hypothetical situations have been considered. For the purpose of this investigation block access counts have been computed for fully specified point and range queries.

Table II shows the results for Point queries for N=100000, and c=100. For point queries, the number of buckets accessed in both structures is naturally 1.0, but the number of block accesses varies. In the grid file, the number of blocks (linear scales plus grid blocks) accessed is simply one more than the number of keys. In the case of k-d-B-tree, this count is about the same for f=1, but increases with the number of cycles used. Therefore the grid file can be considered faster for point queries.

Table III and Table IV give the results for the range queries. The effects of database size, N, and dimension, k, can be observed from the results in Table III. These results have also been plotted in Figures 5 through 9. It is noticed that there is an increased influence of k for larger N. All access counts, except the node accesses in k-d-B-tree, decrease with k. It is interesting to note that the block accesses in grid file decrease with k while node access count increases with k for

f=1. Based on these results, it is concluded that k-d-B-tree structures allow a faster access than grid file for small k cases while the opposite is true for larger k. The value of k where the changeover occurs changes with the database size; for the range covered, it is approximately 8.

Comparison of the bucket access counts indicates a decrease with k for both structures. For the entire N and k ranges covered in these exercises, grid file performs better than a one-cycle k-d-B-tree, with about 20% to 40% fewer access counts. It is desirable to repeat the comparison with multi-cycle k-d-B-trees. This is done next.

In Table IV, and Figures 10 and 11, the effect of the number of cycles, f, and dimension, k, for a file size, N=100,000 can be observed. In a k-d-B-tree structure f may be larger than 1, but how much larger depends on the policy decisions made in a particular implementation. Therefore an f range of 1-4 has been considered. It is clearly observed that the advantages of k-d-B-tree tend to disappear very fast with increased number of cycles. It is interesting to see the variation with number of cycles is steeper in the node access counts and slower in bucket access counts. One might conclude, therefore, that it is advantageous for the range query performance of a k-d-B-tree to make such policy decisions that will lead to smaller number of cycles.

In Table IV the effect of file size is investigated. The results plotted in Figures 12 and 13 show the effects of dimension and the way file size, N, influences these effects. Again, the number of blocks accessed in the grid file case decreases with k while the corresponding number, the number of node accesses, in a k-d-B-tree increases with k. The general nature of these variations is not affected by the file size,

but it is clear that the rate of decrease with k becomes sharper as N increases. In the case of k-d-B-tree, however, the rate of change of the access counts does not seem to be affected appreciably with the file size.

Figures 14 through 16 show the trends of the two structures in FR type queries as the file size increases over wide ranges. It is observed that the performance of the two are roughly parallel in terms of block accesses (when the linear scale plus grid blocks in grid file case, and node accesses in k-d-B-tree case are compared). For the conditions assumed in theses exercises, grid file is clearly the "winner." In terms of bucket accesses, k-d-B-tree seems to be more economical at low dimensions, but the difference decreases at higher dimensions; the two are the same at about k=8. Another interesting observation that can be made in these figures is that the rate of increase of node accesses with N in k-d-B-tree is very slow (about 1/2 of grid file). This is a factor that would make k-d-B-tree more economical for very large databases. For f>1 we have seen earlier that k-d-B-tree access counts increase significantly with f, and therefore these advantages may disappear in a realistic implementation.

These conclusions on the comparative performance of the two files have been included to exemplify the usage of the general formulas presented in this chapter. The actual performances may depend on the parameters not considered in this evaluation.

Table V gives the summary of the formulas that are derived to calculate the block accesses for all query types.

TABLE II

COMPARISON OF BLOCK ACCESSES: FP QUERIES
(N= 100000, c= 100)

	keys	nodes-GF	nodes-KDB	buckets-GF	buckets-KDB
f= 1					
	2	3	2	1	1
	3	4	3	1	1
	4	5	4	1	1
	5	6	5	1	1
	6	7	6	1	1
	7	8	7	1	1
	8	9	8	1	1
	9	10	9	1	1
	10	11	10	1	1
f= 2					
	keys	nodes-GF	nodes-KDB	buckets-GF	buckets-KDB
	2	3	4	1	1
	3	4	6	1	1
	4	5	8	1	1
	5	6	10	1	1
	6	7	12	1	1
	7	8	14	1	1
	8	9	16	1	1
	9	10	18	1	1
	10	11	20	1	1
f= 3					
	keys	nodes-GF	nodes-KDB	buckets-GF	buckets-KDB
	2	3	6	1	1
	3	4	9	1	1
	4	5	12	1	1
	5	6	15	1	1
	6	7	18	1	1
	7	8	21	1	1
	8	9	24	1	1
	9	10	27	1	1
	10	11	30	1	1
f= 4					
	keys	nodes-GF	nodes-KDB	buckets-GF	buckets-KDB
	2	3	8	1	1
	3	4	12	1	1
	4	5	16	1	1
	5	6	20	1	1
	6	7	24	1	1
	7	8	28	1	1
	8	9	32	1	1
	9	10	36	1	1
	10	11	40	1	1

TABLE III

COMPARISON OF BLOCK ACCESSES: FR QUERIES
(N= 100000, c= 100)

	keys	nodes-GF	nodes-KDB	buckets-GF	buckets-KDB
f= 1					
	2	331.92	15.37	164.96	206.42
	3	191.06	29.91	94.03	117.66
	4	120.43	39.50	58.22	79.41
	5	98.15	48.32	46.57	63.53
	6	80.52	53.88	37.26	50.82
	7	66.61	56.84	29.81	40.66
	8	55.69	57.81	23.85	32.53
	9	47.15	57.34	19.08	26.02
	10	40.52	55.87	15.26	20.82
f= 2					
	keys	nodes-GF	nodes-KDB	buckets-GF	buckets-KDB
	2	331.92	69.83	164.96	363.80
	3	191.06	122.83	94.03	291.04
	4	120.43	157.29	58.22	232.83
	5	98.15	175.25	46.57	186.26
	6	80.52	180.90	37.26	149.01
	7	66.61	178.03	29.81	119.21
	8	55.69	169.65	23.85	95.37
	9	47.15	157.99	19.08	76.29
	10	40.52	144.65	15.26	61.04
f= 3					
	keys	nodes-GF	nodes-KDB	buckets-GF	buckets-KDB
	2	331.92	214.52	164.96	520.70
	3	191.06	327.55	94.03	416.56
	4	120.43	392.40	58.22	333.24
	5	98.15	420.29	46.57	266.60
	6	80.52	422.36	37.26	213.28
	7	66.61	407.44	29.81	170.62
	8	55.69	382.06	23.85	136.50
	9	47.15	350.93	19.08	109.20
	10	40.52	317.32	15.26	87.36
f= 4					
	keys	nodes-GF	nodes-KDB	buckets-GF	buckets-KDB
	2	331.92	408.74	164.96	622.94
	3	191.06	584.00	94.03	498.35
	4	120.43	677.99	58.22	398.68
	5	98.15	712.92	46.57	318.94
	6	80.52	707.69	37.26	255.16
	7	66.61	676.61	29.81	204.12
	8	55.69	630.07	23.85	163.30
	9	47.15	575.44	19.08	130.64
	10	40.52	517.76	15.26	104.51

TABLE IV

COMPARISON OF BLOCK ACCESSES: FR QUERIES
(f= 1, c= 100)

	keys	nodes-GF	nodes-KDB	buckets-GF	buckets-KDB
N= 10000					
	2	46.51	6.28	22.25	27.85
	3	31.38	10.89	14.19	19.35
	4	26.70	14.72	11.35	15.48
	5	23.16	17.41	9.08	12.39
	6	20.53	19.14	7.26	9.91
	7	18.62	20.13	5.81	7.93
	8	17.30	20.57	4.65	6.34
	9	16.44	20.60	3.72	5.07
	10	15.95	20.34	2.98	4.06
N= 30000					
	keys	nodes-GF	nodes-KDB	buckets-GF	buckets-KDB
	2	117.75	9.51	57.87	72.42
	3	68.98	16.61	32.99	42.22
	4	53.53	23.23	24.76	33.78
	5	44.62	27.88	19.81	27.02
	6	37.70	30.82	15.85	21.62
	7	32.36	32.42	12.68	17.29
	8	28.29	33.02	10.14	13.84
	9	25.23	32.88	8.11	11.07
	10	22.98	32.23	6.49	8.85
N= 100000					
	keys	nodes-GF	nodes-KDB	buckets-GF	buckets-KDB
	2	331.92	15.37	164.96	206.42
	3	191.06	29.91	94.03	117.66
	4	120.43	39.50	58.22	79.41
	5	98.15	48.32	46.57	63.53
	6	80.52	53.88	37.26	50.82
	7	66.61	56.84	29.81	40.66
	8	55.69	57.81	23.85	32.53
	9	47.15	57.34	19.08	26.02
	10	40.52	55.87	15.26	20.82
N= 300000					
	keys	nodes-GF	nodes-KDB	buckets-GF	buckets-KDB
	2	860.04	24.17	429.02	536.85
	3	492.08	53.15	244.54	306.01
	4	282.78	65.54	139.39	173.23
	5	208.20	81.83	101.60	138.59
	6	168.56	92.19	81.28	110.87
	7	137.05	97.67	65.02	88.69
	8	112.04	99.42	52.02	70.96
	9	92.23	98.46	41.61	56.76
	10	76.58	95.59	33.29	45.41

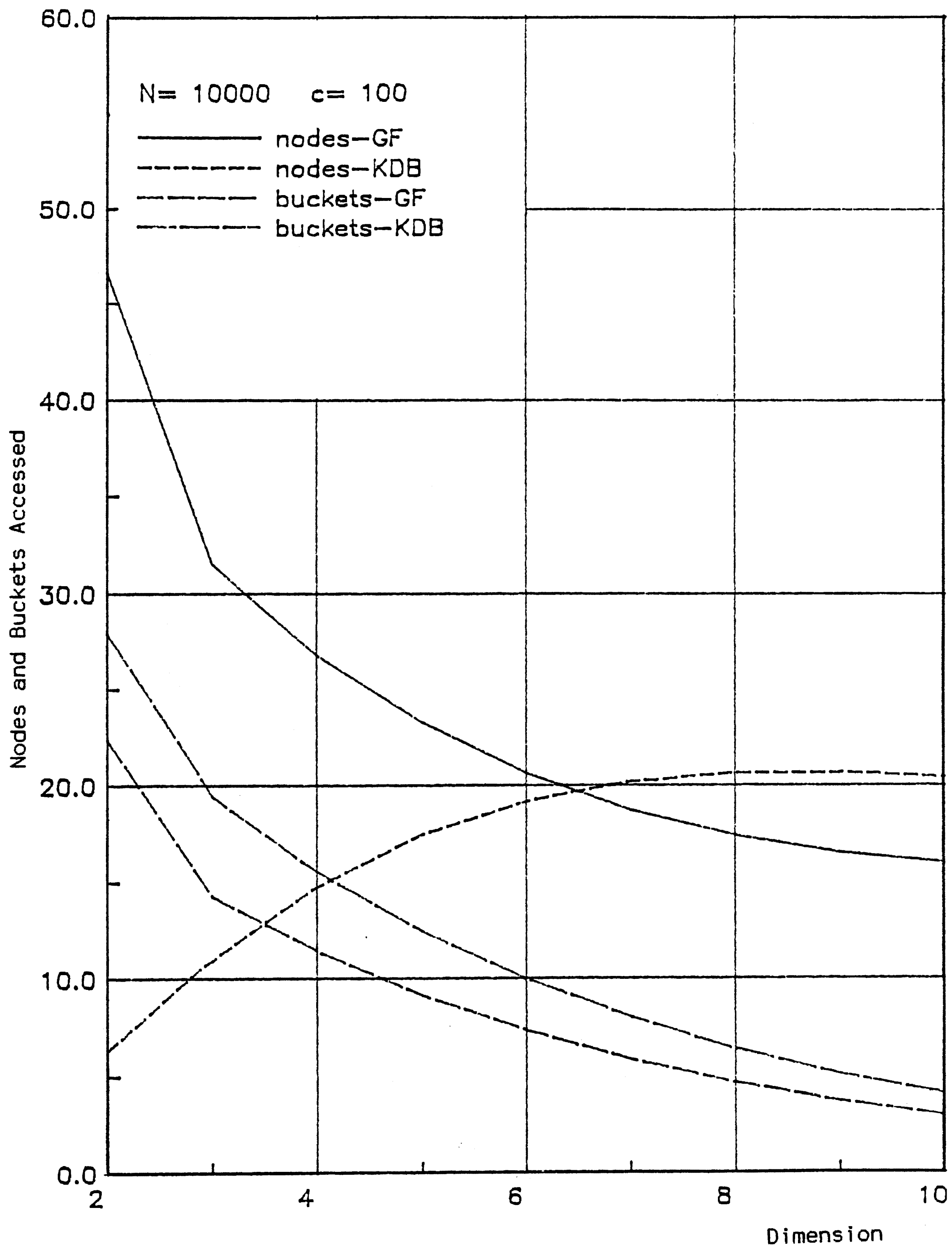


Figure 5. Comparison of Block Accesses: FR Case
(Database Size 10000)

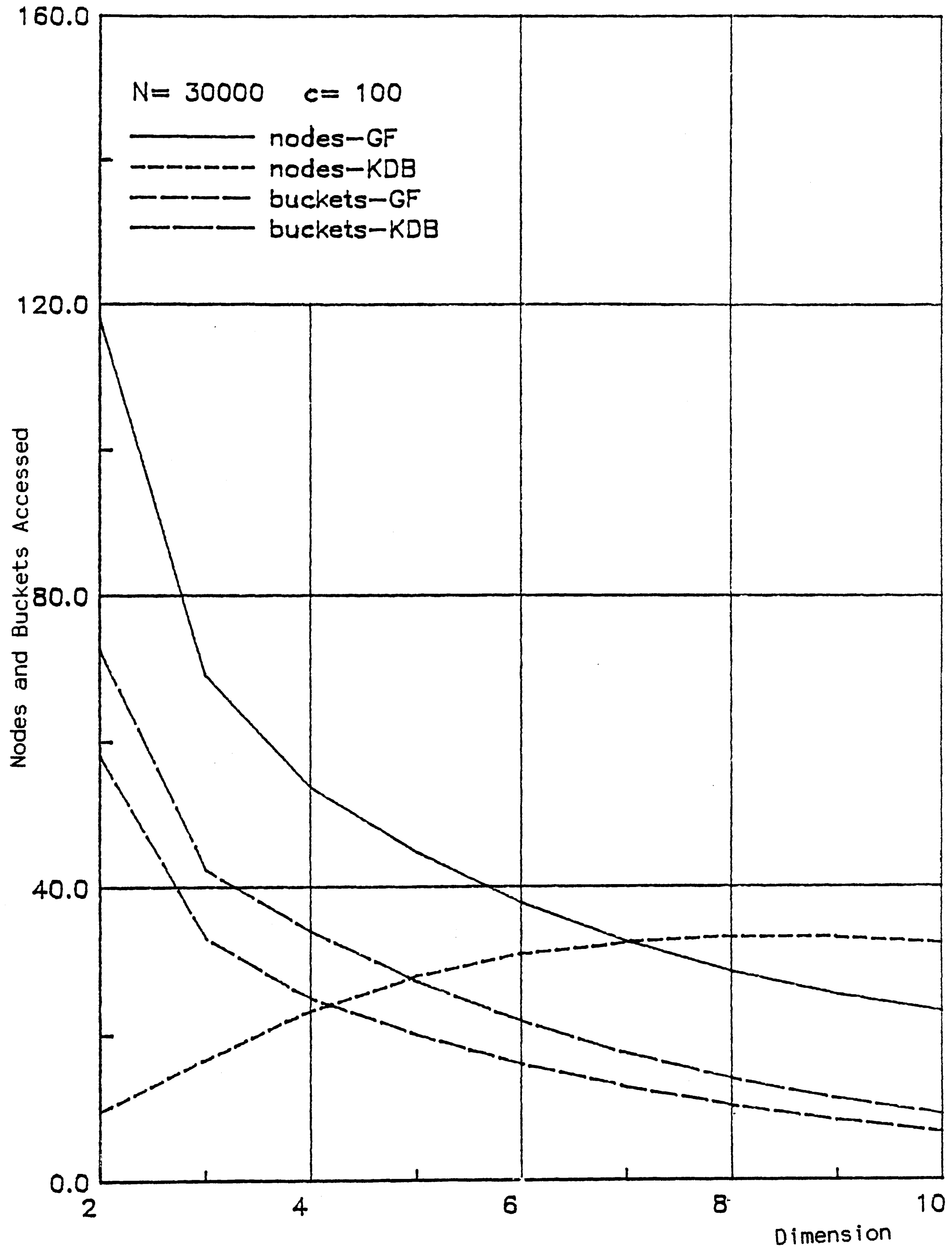


Figure 6. Comparison of Block Accesses: FR Case
(Database Size 30000)

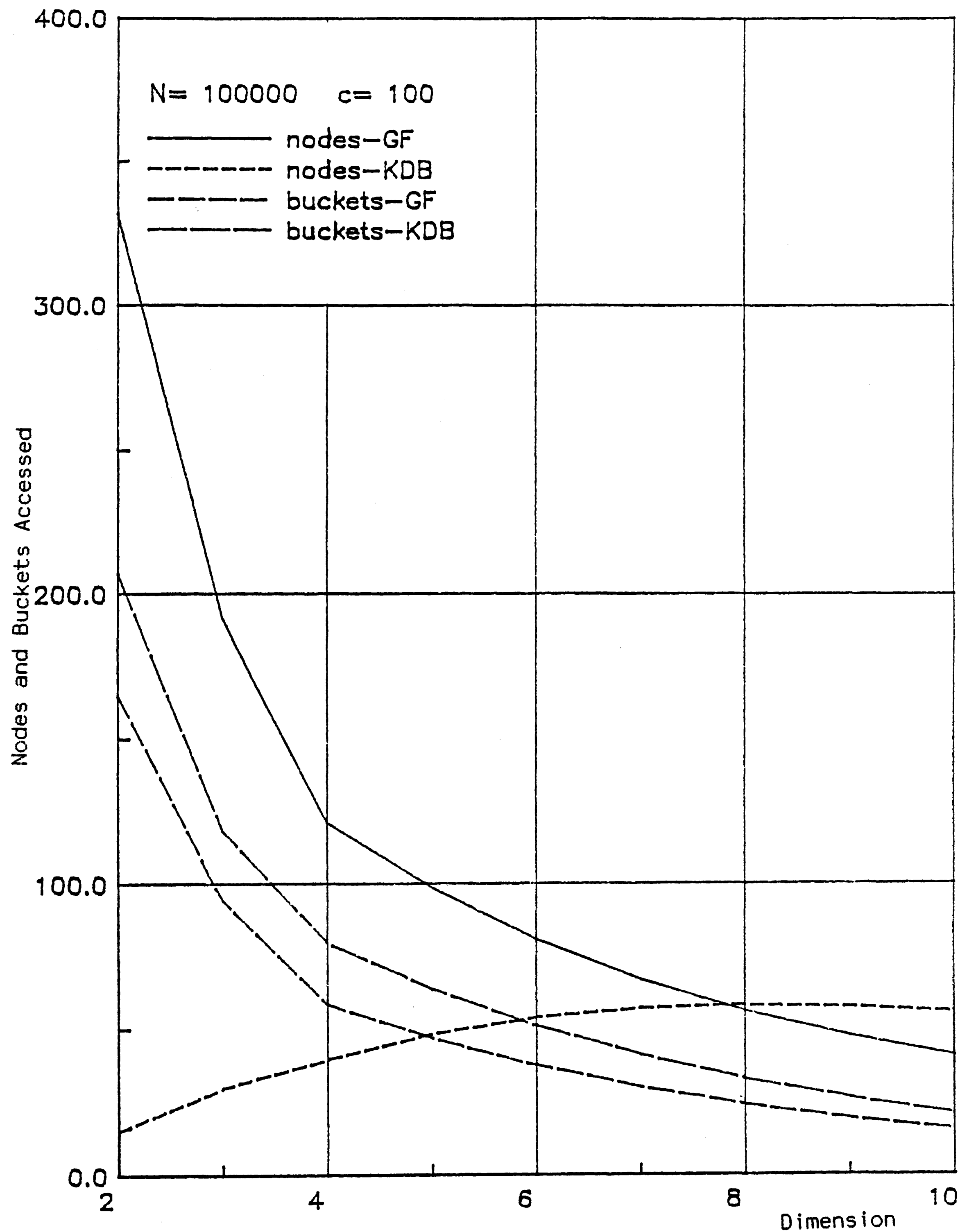


Figure 7. Comparison of Block Accesses: FR Case
(Database Size 100000)

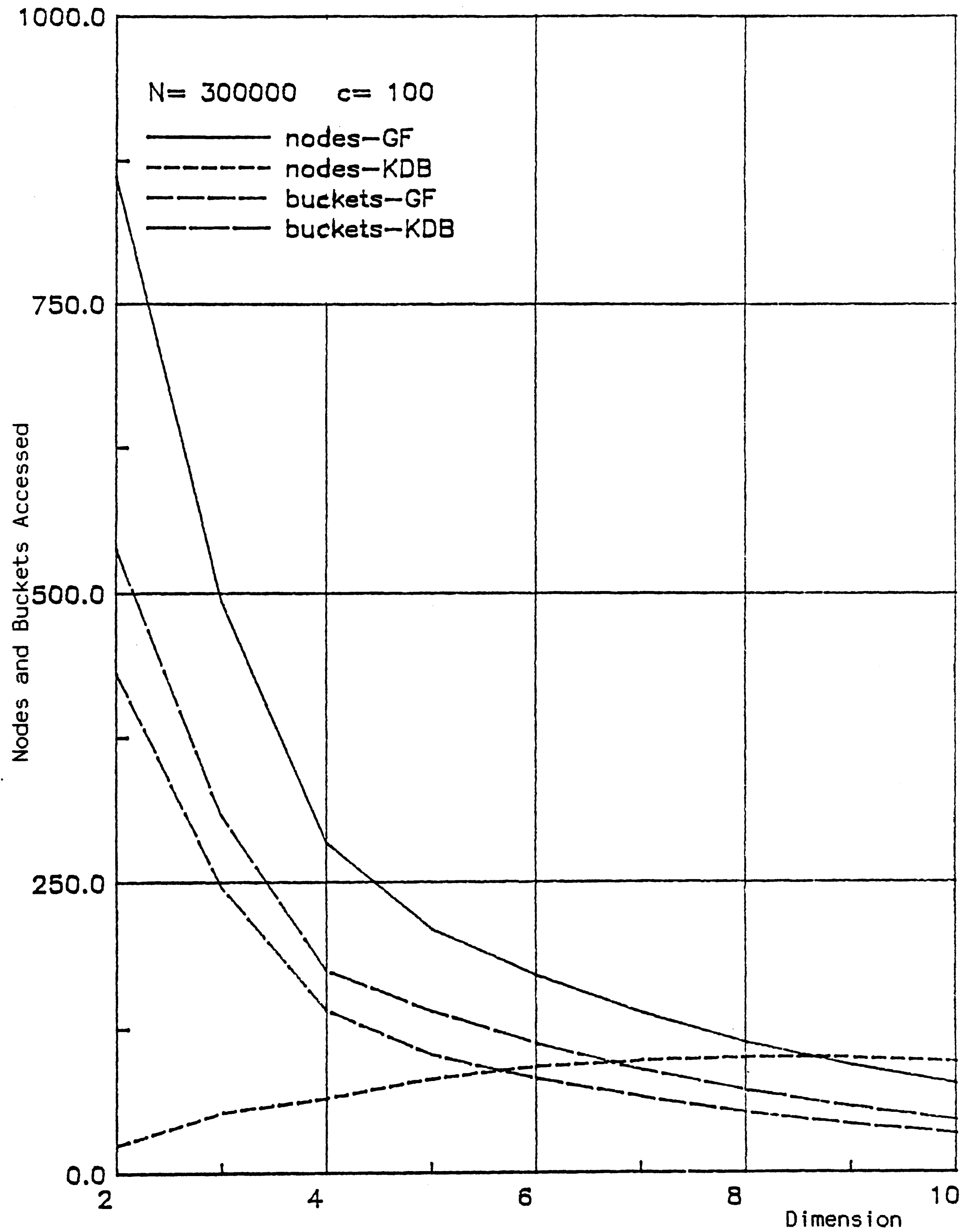


Figure 8. Comparison of Block Accesses: FR Case (Database Size 300000)

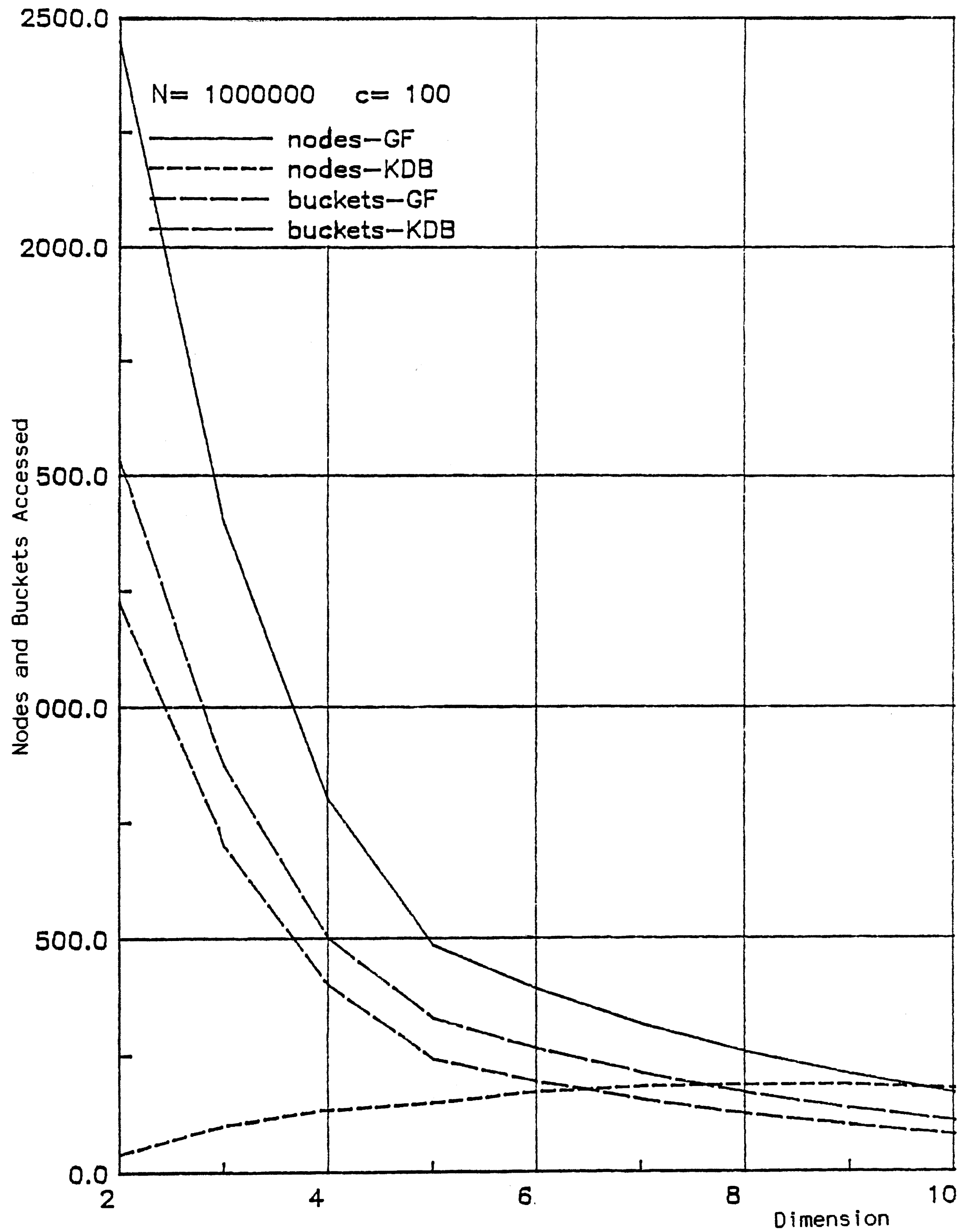


Figure 9. Comparison of Block Accesses: FR Case
(Database Size 1000000)

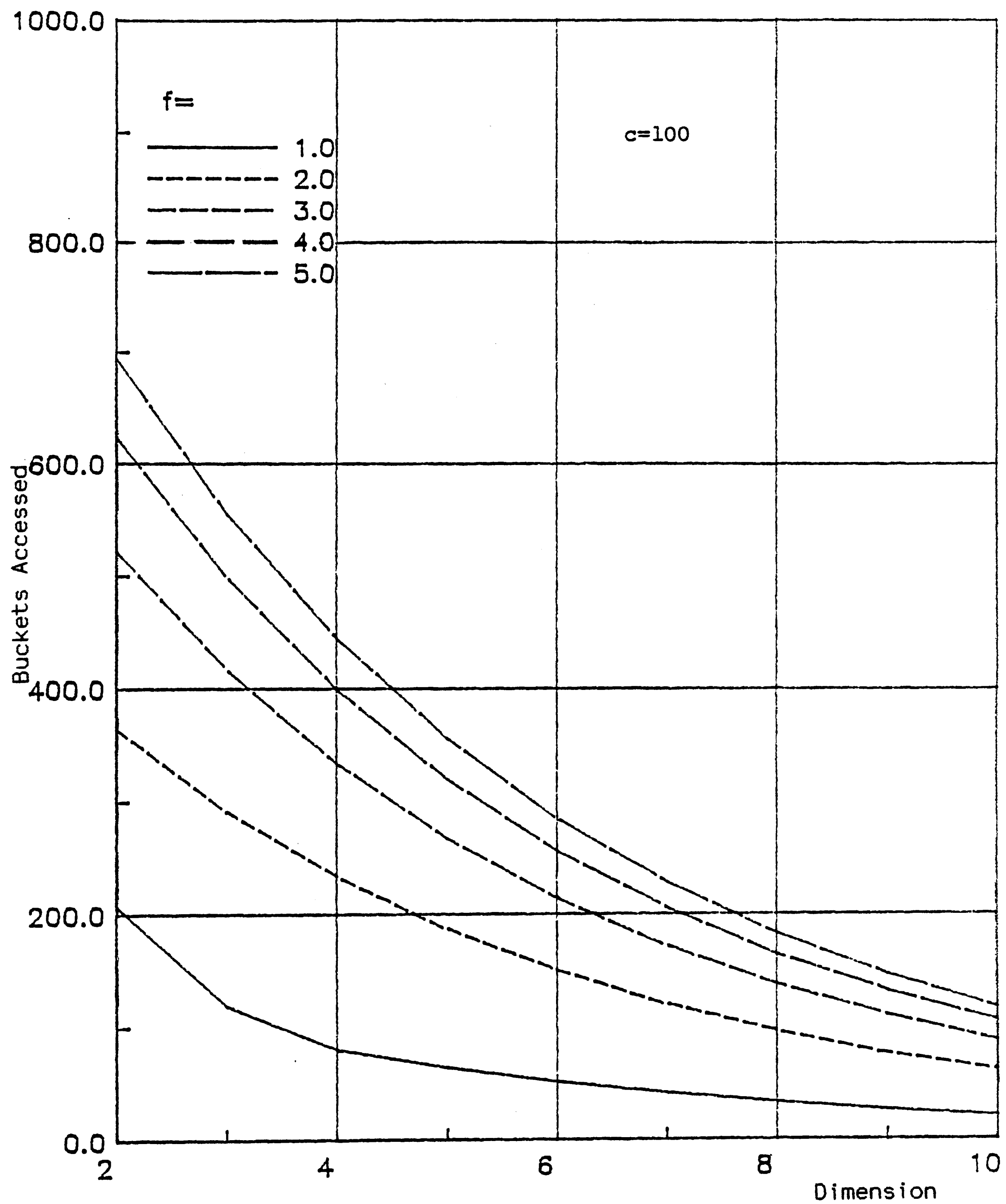


Figure 10. Effect of Cycles on k-d-B-tree Node Accesses (Database Size 100000)

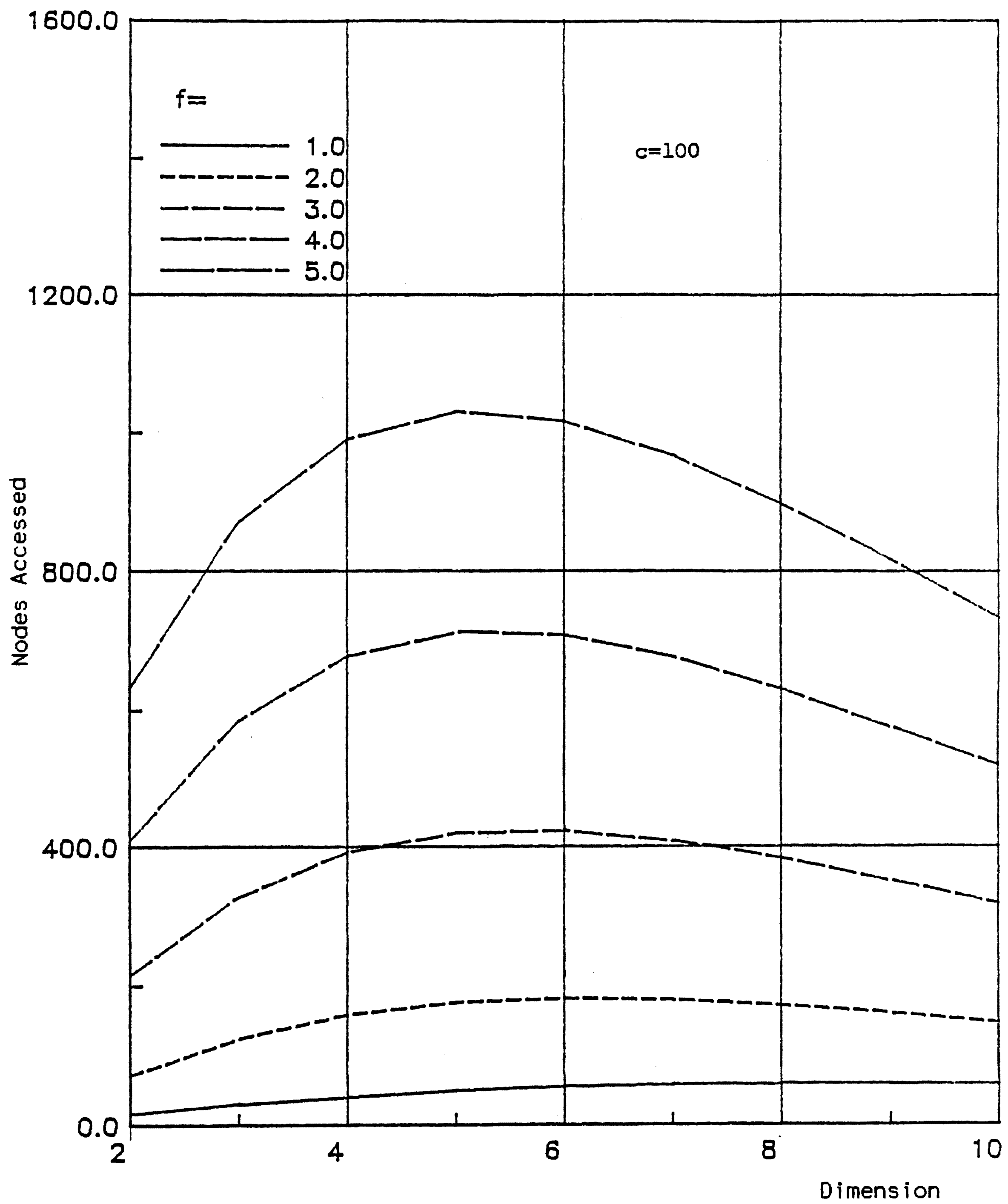


Figure 11. Effect of Cycles on k-d-B-tree Bucket Accesses (Database Size 100000)

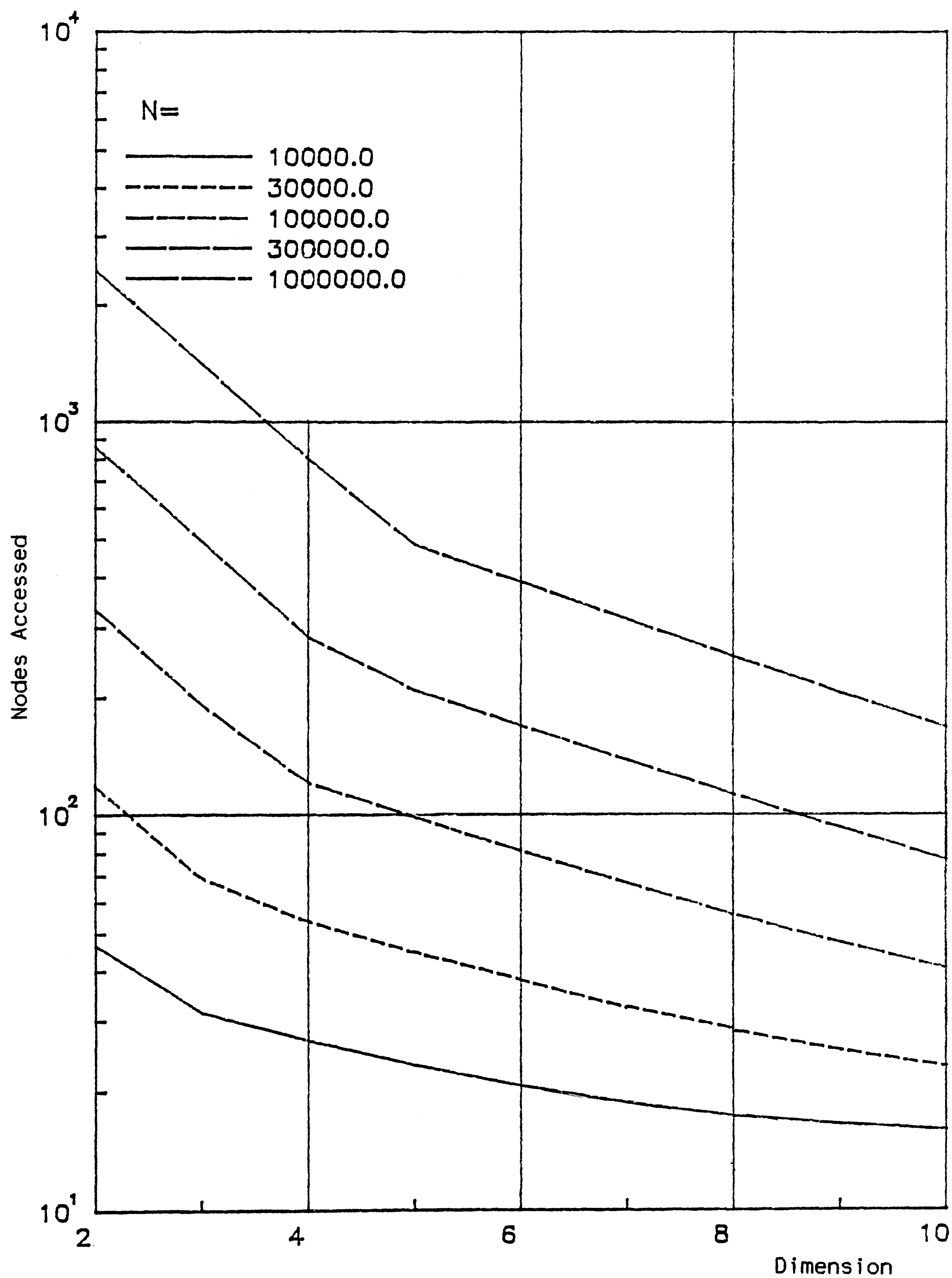


Figure 12. Effect of Dimension on Grid File Node Accesses

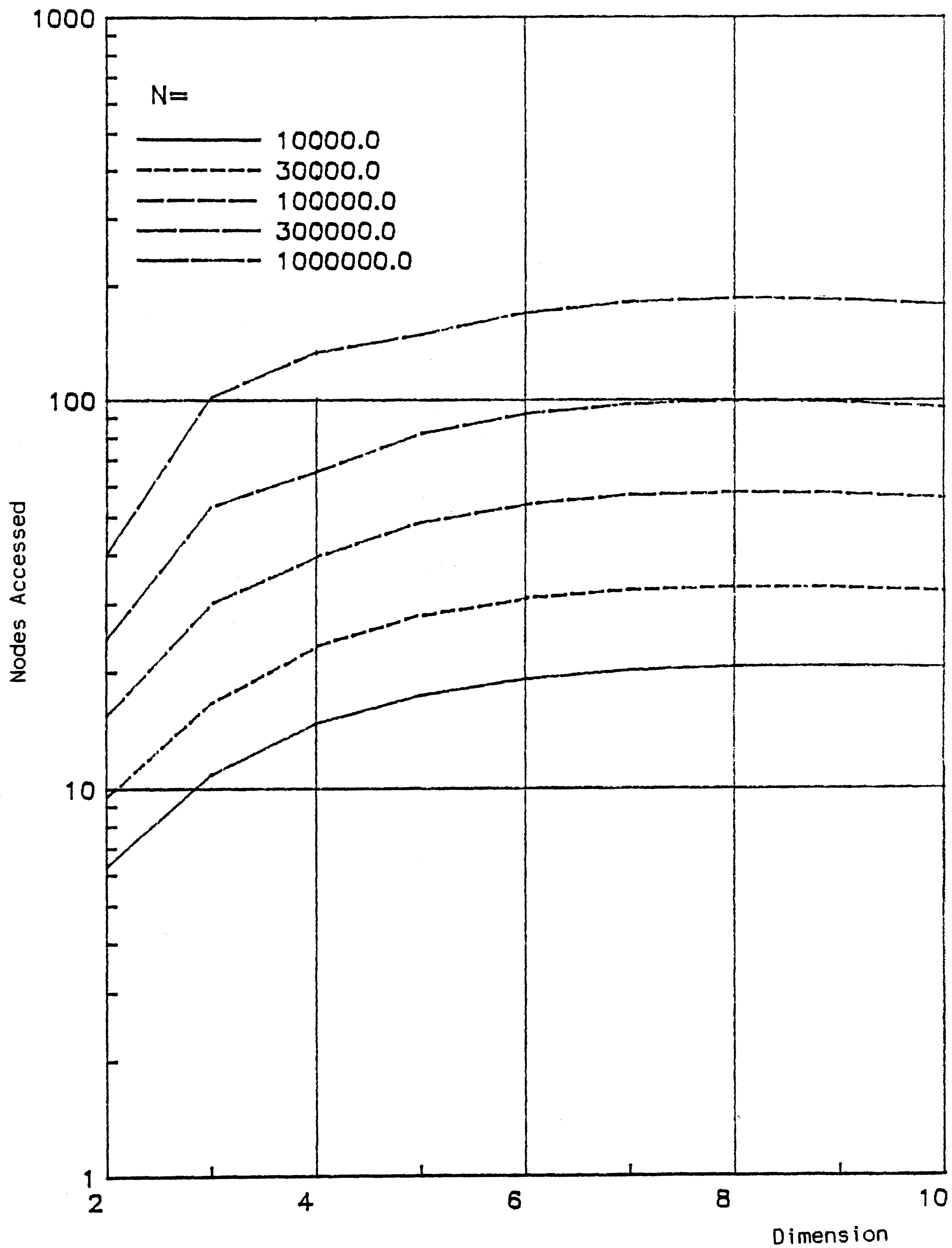


Figure 13. Effect of Dimension on k-d-B-tree Node Accesses

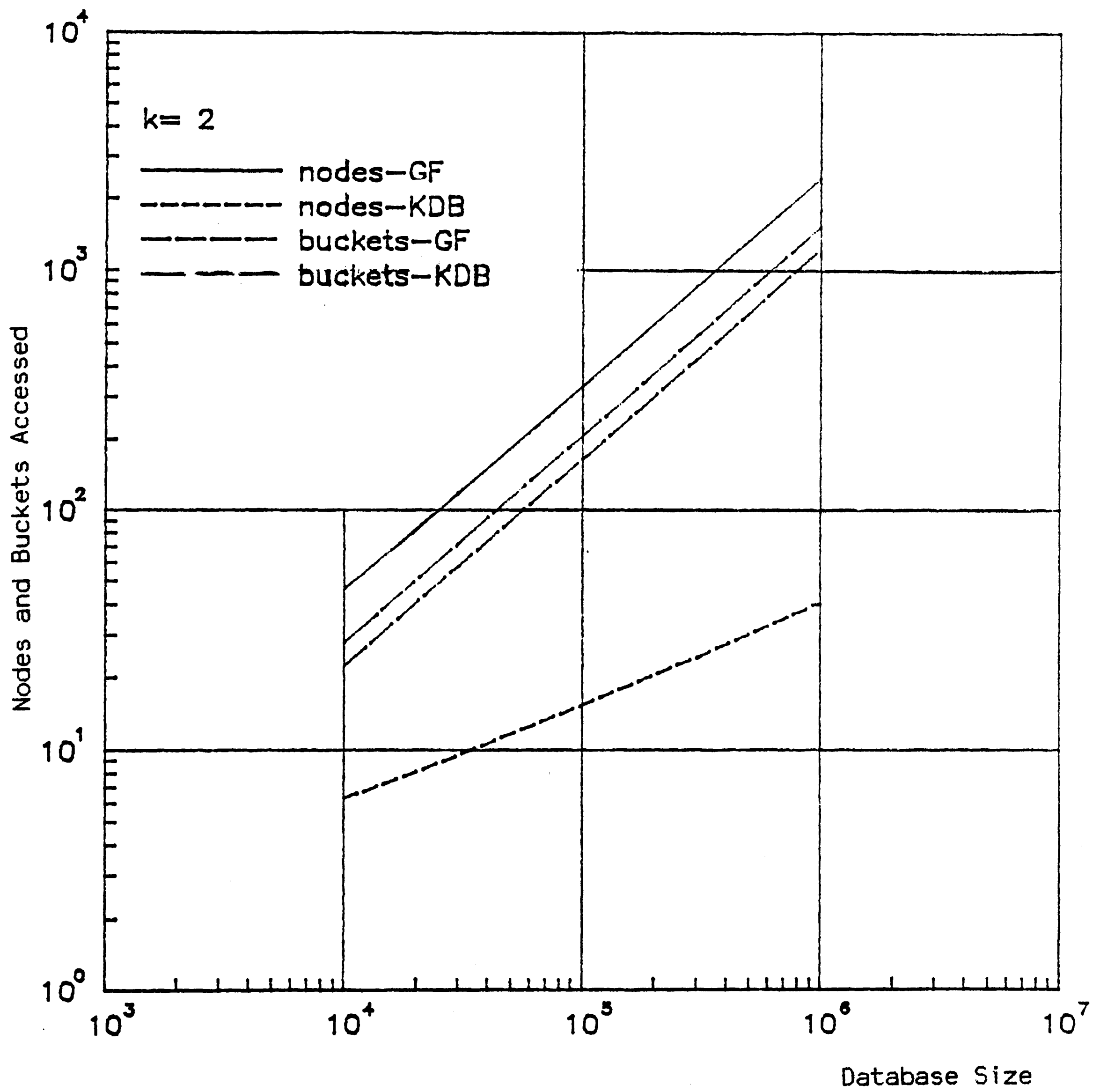


Figure 14. Comparison of Block Accesses: FR Case (Dimension 2)

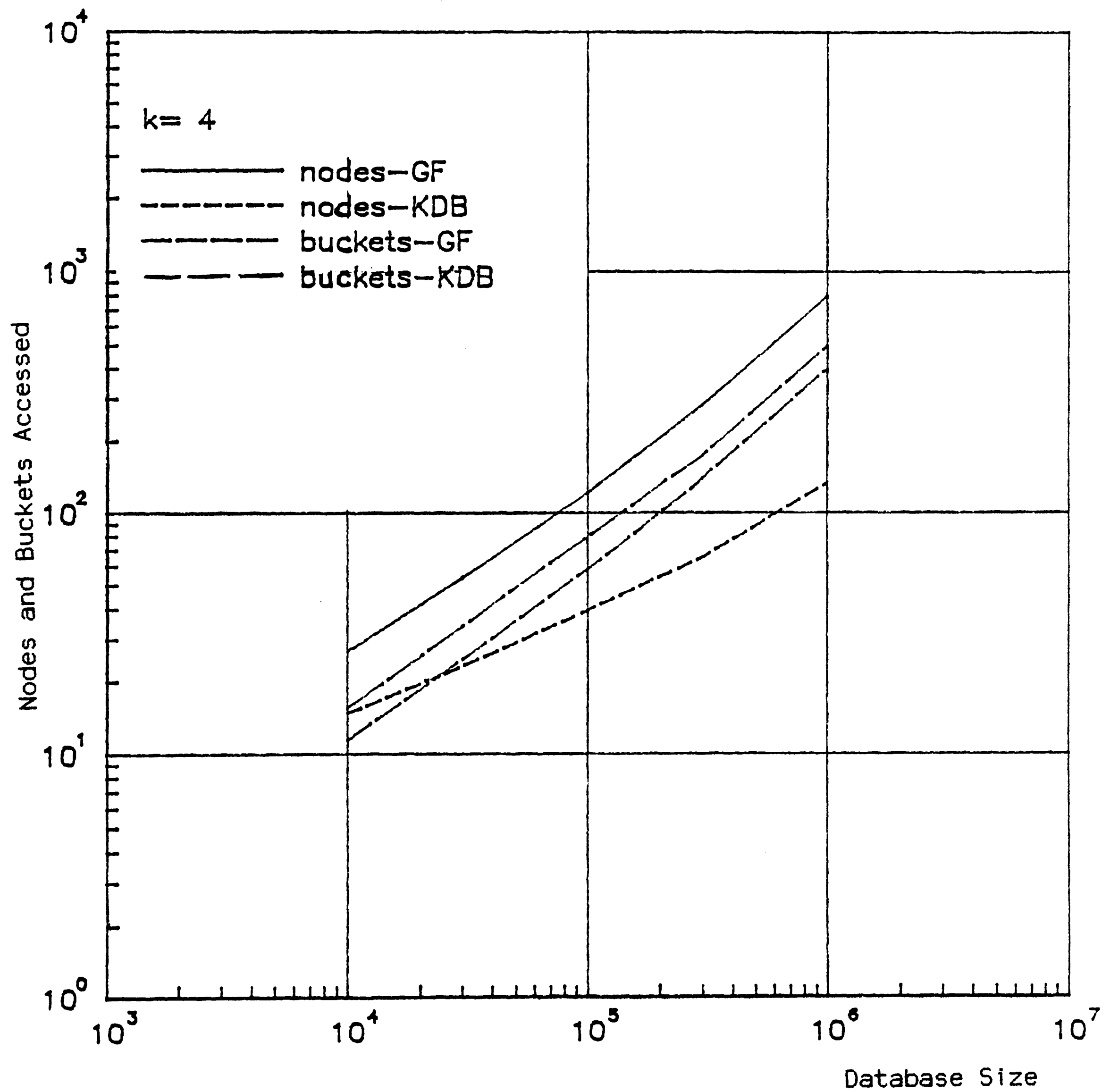


Figure 15. Comparison of Block Accesses: FR Case (Dimension 4)

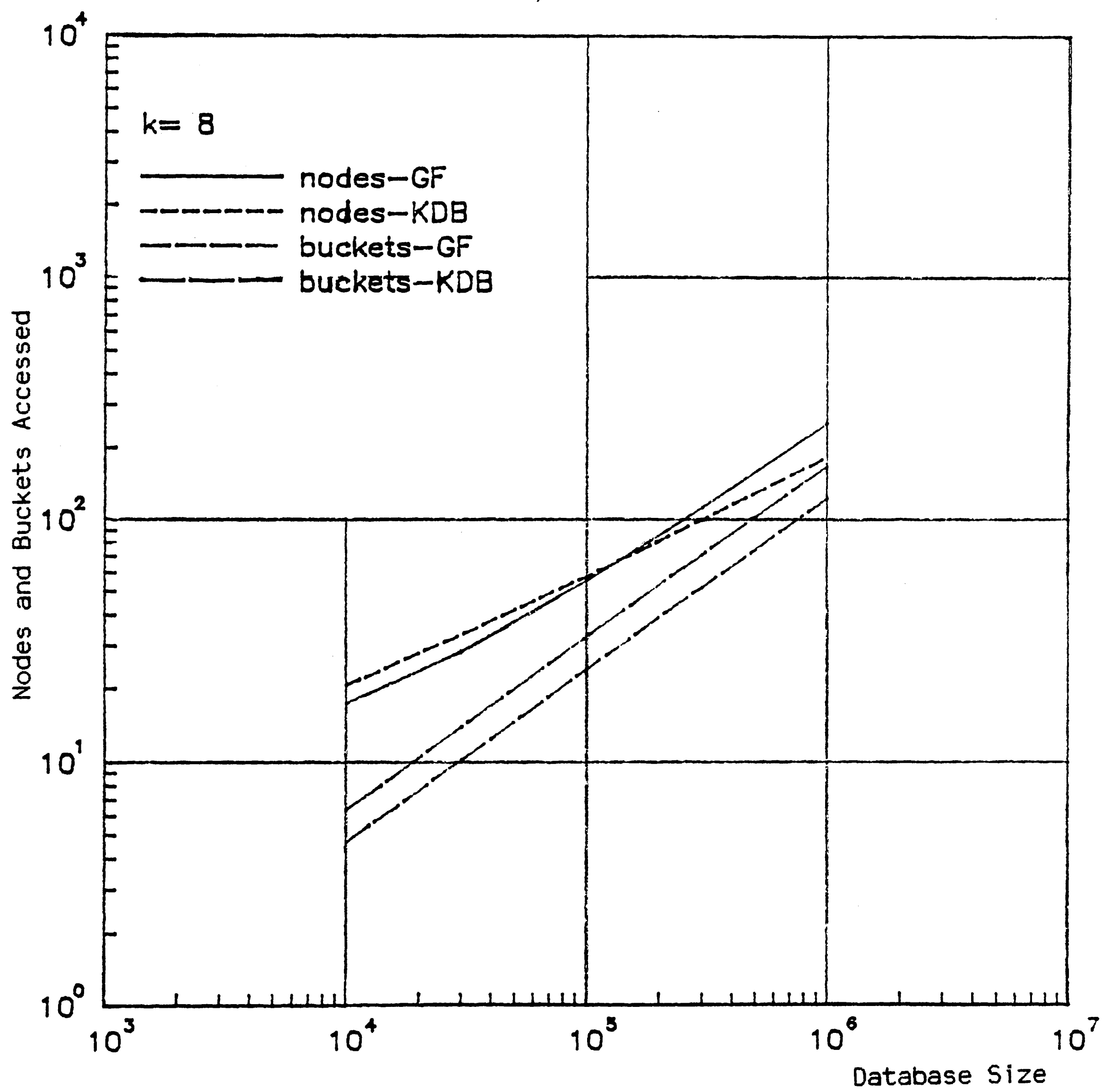


Figure 16. Comparison of Block Accesses: FR Case (Dimension 8)

TABLE V

SUMMARY OF BLOCK ACCESS FORMULAS FOR QUERIES

Grid File	
FP Case:	$a_l = k$
	$a_g = 1$
	$a_b = 1$
PP Case:	$a_l = k_s$
	$a_g = n^{(k-k_s)}$
	$a_b = n^{(k-k_s)} / r$
FR Case:	$a_l = k$
	$a_g = n_a^k$
	$a_b = n_a^k / r$
PR Case:	$a_l = k_s$
	$a_g = n_a^{k_s} \; n^{(k-k_s)}$
	$a_b = n_a^{k_s} \; n^{(k-k_s)} / r$
FF Case:	$a_l = k$
	$a_g = n^k$
	$a_b = n^k / r$
PF Case:	$a_l = k_s$
	$a_g = n^k$
	$a_b = n^k / r$

TABLE V (Continued)

K-d-B-tree

FP Case:

$$a_n = fk$$

$$a_b = 1$$

PP Case:

$$a_n = 1 + \sum_{i=1}^{k-1} m^{i(1-k_s/k)} + \sum_{j=1}^{f-1} \sum_{i=1}^{k} m^{(jk-i)}$$

$$a_b = m^{(k-k_s)} \; m^{(f-1)k}$$

FR Case:

$$a_n = 1 + \sum_{i=1}^{k-1} m_a{}^{i} + m_a{}^{k} \sum_{j=1}^{f-1} \sum_{i=1}^{k} m^{(jk-i)}$$

$$a_b = m_a{}^{k} \; m^{(f-1)k}$$

PR Case:

$$a_n = 1 + \sum_{i=1}^{k-1} m_a{}^{i(k_s/k)} \; m^{i(1-k_s/k)} + m_a{}^{k} \sum_{j=1}^{f-1} \sum_{i=1}^{k} m^{(jk-i)}$$

$$a_b = m_a{}^{k_s} \; m^{(k-k_s)} \; m^{(f-1)k}$$

FF Case:

$$a_n = 1 + \sum_{i=1}^{fk-1} m^{i} = (m^{fk}-1) / (m-1)$$

$$a_b = m^{(fk)}$$

PF Case:

$$a_n = 1 + \sum_{i=1}^{fk-1} m^{i} = (m^{fk}-1) / (m-1)$$

$$a_b = m^{(fk)}$$

CHAPTER V

INSERTION PERFORMANCE

Introduction

Insertion is an important issue affecting the performance of any multikey access mechanism. Since their nature is different, the grid file and the k-d-B-tree act differently with respect to insertions. In this chapter our attention shall focus on the relative performance of the two structures from this point of view.

In general, insertion includes three steps. The first step needed is to find the right location to store the new record. Since the new record is completely specified (all keys have some values), a point query search will give the right bucket address for the new record. The second step involves checking the space availability in the bucket to decide about whether or not a split is needed. If there is enough room in the bucket, then the record is simply added to that bucket. If the bucket is full, then that bucket has to be split in two. This will also cause some reorganization in the structure. This process will be called "splitting" in the following. The final step in the insertion process is actually storing the new record in the bucket it belongs to, which means an actual disk write.

The cost associated with each step can be different for two cases of insertions, insertions involving splitting and insertions without splitting. The cost of the first step is the cost of a point query that

is calculated in Chapter III and is same for both cases. The cost of the final step is the cost of an actual disk write operation that depends on hardware but is also the same for both cases. In any case the new record needs to be correctly located and stored. The difference comes from the second step that checks whether or not splitting is required. If splitting is not required then the cost of the second step is zero. If splitting is necessary then the cost of the insertion can be considerable. Therefore this requires further examination.

Splitting involves a division of the full bucket that is the target of the new insertion. But it also creates an overhead related with the access mechanism itself. After each bucket split, both grid file and k-d-B-tree structures need to be reorganized. The cost of this reorganization overhead should be considered in estimating the average cost of an insertion.

Measuring the Cost of an Insertion

To measure the cost of an insertion, the following probability formula that has been introduced in Chapter III will be used.

$$E(\text{insertion}) = C1\ P(x) + C2\ (1 - P(x)) \qquad (5.1)$$

where E(insertion) is the expected cost of an insertion, C1 is the cost of an insertion with splitting, and C2 is the cost of an insertion without splitting. P(x) represents the probability of occurrence of splitting at an insertion, and 1 - P(x) the probability that splitting will not occur.

Since splitting is rare we may expect to have a small probability value for P(x). On the other hand, splitting is a long operation of reorganization of the entire structure, therefore the cost of insertion

with splitting, C1, should be expected to be much larger than that without splitting, C2. C1 and C2 can be considered as the maximum and the minimum costs for the insertion respectively. Each of these elements will be examined in the following.

Probability of Splitting

We know that we have to do a splitting in both grid files and k-d-B-trees when a bucket is full and a new record to be inserted falls into that bucket. The important question here is what the probability, P(x), of this happening is, since the expected cost depends on this probability. Now we need to quantify P(x).

As the following analysis will indicate, the occurrence of splitting closely interacts with the bucket occupancy ratio that the structure maintains. In grid file and k-d-B-tree structures there is a set of buckets (b). When the file matures, a certain occupancy ratio is maintained. Afterwards, insertions will, from time to time, cause splitting. In a matured file the splitting will come at a certain period in a probabilistic sense. This period can be measured in terms of average number of records that can be inserted without necessitating a split and can be called "return period of splitting (T_R)".

The return period, if it can be calculated, is related to the probability of splitting. For example, if T_R = 10, that is, a splitting occurs every tenth insertion on the average, then the probability that splitting will occur, P(x), in any one insertion is 0.1, thus,

$$P(x) = 1 / T_R \qquad (5.2)$$

Therefore we need to evaluate T first. In order to find T, consider a hypothetical set of ten (b=10) buckets each of which can hold

a one hundred (c=30) records. Since we assume that the database is uniform, the probability that a new record will "hit" a given bucket is 0.1 in this case. Now if we assume a key value range of 0 to 1, then the first bucket will take records in the range (0, 0.1), the next between 0.1 and 0.2 and so on, until bucket number 10 which will take records in the range (0.9, 1). The total available capacity will be bc=300, and at maturity there will be bce=300e records where e is the average occupancy ratio. To study how the buckets are filled, we can generate uniformly distributed random numbers, between 0 and 1, and "put" each record into its bucket according to the value of the random number, and continue this until one bucket overflows. This way we will see both the average occupancy ratio, e, and its distribution over the buckets at the moment splitting occurs.

When this numeric experiment was done, and repeated 100 times for accuracy, the results plotted in Figure 17 were obtained. It is seen that the average bucket occupancy ratio is e = 0.77, and its standard deviation is sigma = 0.064.

The most significant result of this numeric experiment is that the occupancy ratio value obtained is the highest value that can be obtained under the conditions assumed, and splitting occurs when this ratio is reached. The relationship of e at splitting, e_split (i.e., e=e_split) to the return period of splitting is established next.

An important point should be clarified at this point. Although we referred to the random numbers generated in the numerical experiments as "key values", as if we are considering a one dimensional structure, this is not necessarily so. Because, irrespective of the number of keys, and no matter which access mechanism is utilized, when we come to the bucket

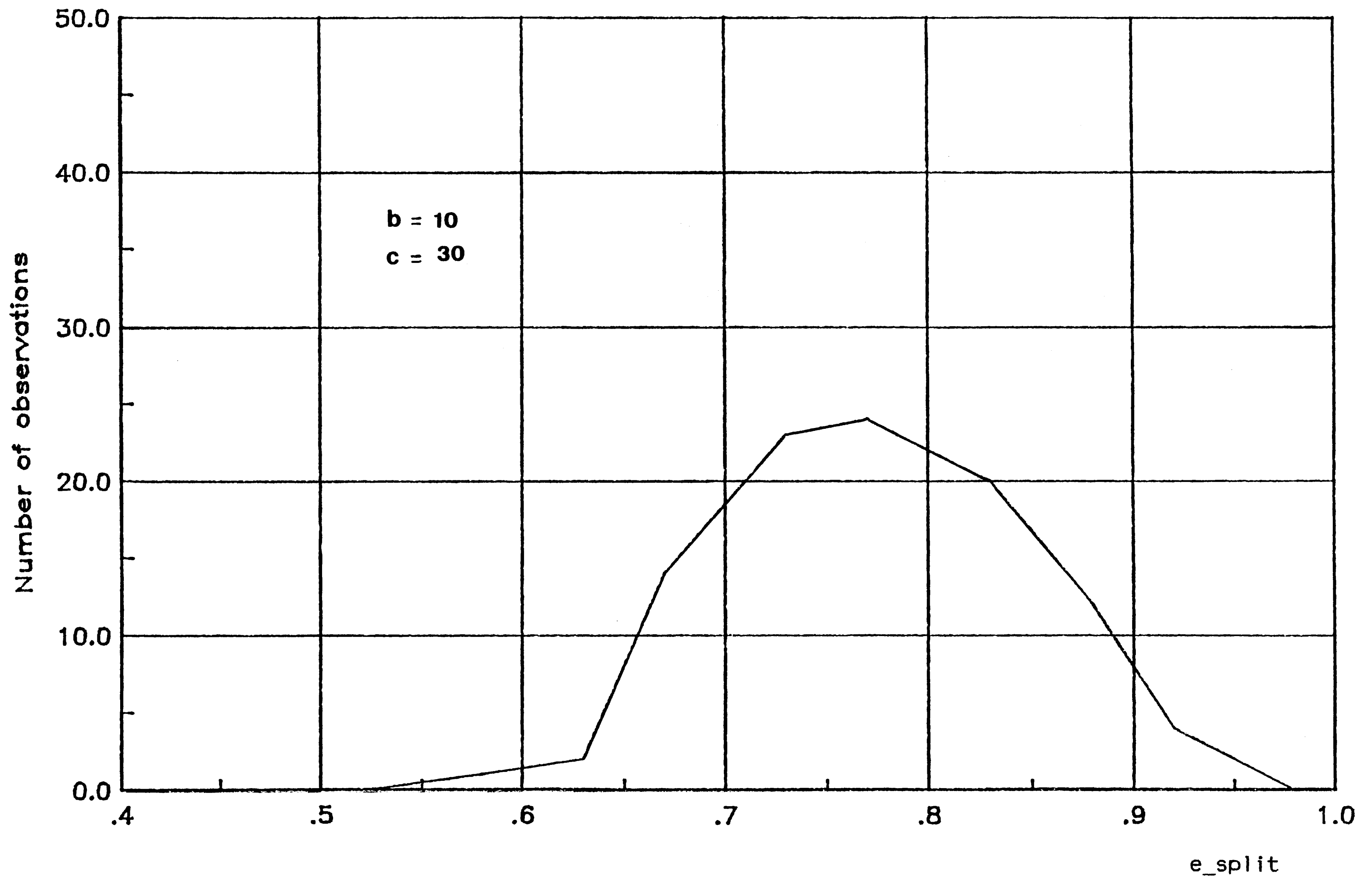


Figure 17. Distribution of Bucket Occupancy Ratio at Splitting

level there is only one array of buckets. What we assume is that the file structure is capable of distributing the records "correctly" to the buckets, such that the probability of hitting any given bucket is always 1/b. Therefore the results are equally valid for any file structure.

These distribution experiments show that, for b=10 and c=30, on the average, a splitting is encountered when a bucket occupancy of .77 is reached (e_split). Suppose a database maintains a .70 mean occupancy ratio (e0). Then we are free to add .07bc records without having a splitting. We call this value the "mean free capacity". In general the mean free capacity is

$$T_R = (e_split - e0) \; b \; c \qquad (5.3)$$

The mean free capacity is the number of records that can be inserted until splitting occurs. If we assume that the system returns back to the mean occupancy ratio after splitting, then obviously the return period of splitting is equal to the mean free capacity. Since the occupancy ratio will probably fluctuate around the mean value, e0, T_R may be taken as an estimate of the return period. Therefore, the probability of splitting upon an insertion can be expressed as

$$P(x) = 1 \; / \; [\; (e_split - e0) \; b \; c \;] \qquad (5.4)$$

As an example of the usage of these results, consider the mean occupancy ratios in implementations of the grid file [19] and the k-d-B-tree [14] which report that they maintained mean occupancy ratios of 0.6 and 0.7, respectively. Using the same values given in the distribution experiments (b=10, c=30, e_split=0.77) the return period of splitting can be calculated (Equation 5.3). As a result of these calculations, the return period of splitting is 21 and 51 records for grid file and k-d-B-tree respectively. That means, on the average, splitting occurs

after every 21 records in grid file whereas it only occurs after every 51 records in k-d-B-tree. For this specific case, splitting is almost 2.5 times more frequent in the grid file structure than in the k-d-B-tree structure.

Since the result found above is very useful, we have repeated the numerical experiments for a wide range of parameters. The details of these experiments are given in Appendix B. Figure 18 shows the summary of the results. It is interesting to see the dependence of e_split on the number and size of the buckets. As the number of buckets increase e_split decreases, and as the bucket capacity increases e_split increases.

Important practical conclusions can be derived from these results. A smaller e_split means that there will be frequent splittings resulting in a large free capacity. The actual splitting period will depend on the mean occupancy ratio maintained by the file structure. If this ratio is kept high then, in general, splitting will occur more frequently. Also, if suitably large bucket capacity is not used, it will not be possible to maintain a high occupancy ratio. It is also clear that a high value of e_split is desirable from the point of view of minimizing splitting. Numerical results indicate that, if this is a critical issue, one has to use a relatively small number of larger capacity buckets to increase e_split and thereby improve insertion performance.

Cost of Splitting

To complete the cost analysis, we need to determine the cost of splitting, C1. There are many factors and situations to consider in

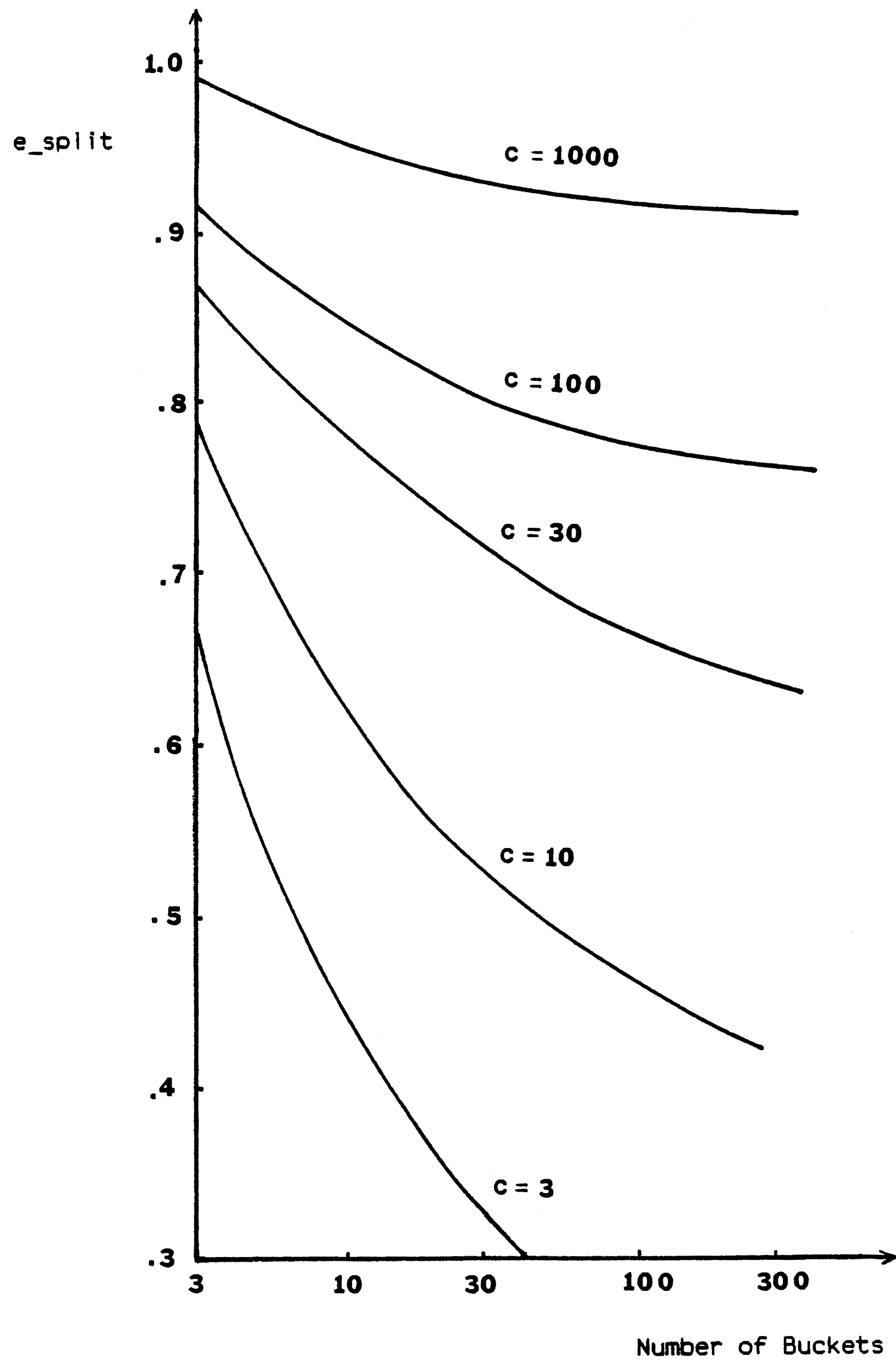


Figure 18. Bucket Occupancy Ratio at Splitting

both the grid file and k-d-B-tree in order to estimate this cost. We will now examine the splitting process in both structures.

For the grid file there can be two situations. The records in the full bucket can belong to several grid blocks in the grid directory. When the new bucket is created, the bucket addresses in the corresponding grid blocks need to be updated to reflect the change. This situation does not affect the grid directory size, it only requires an update. But, if the records in the full bucket belong to a single grid block in the grid directory then the situation becomes more complex. The grid directory needs to be reorganized to hold the newly created bucket address. This reorganization cause the splitting of some grid blocks, changing the grid directory size. Depending on policy decisions, this size change can as much as double the grid directory size. Different splitting policies result in different refinements of the grid partitions. Choice of dimension and location (the point at which the linear scale is partitioned) is important.

The simplest splitting policies choose the dimension according to a fixed schedule, perhaps cyclically. A splitting policy may favor keys by splitting the corresponding dimension(s) more often than others. This has the effect of increasing the precision of answers to partially specified queries in which the favored key(s) is specified, but others are not. For simplicity, the location of a split may be chosen as the midpoint of the related interval for the simplicity. But according to studies [19], little is changed if the splitting point is chosen from a set of values that are convenient for a given application.

For the k-d-B-tree there are also two cases similar to the ones in the grid file. If the leaf node partitioning related with the full

bucket have enough room available, then the newly created bucket can only cause the update of that leaf node. But if the leaf node is too full to accommodate the newly created bucket then a change in the partitioning is needed. The important point here is that a number of levels are involved in the reorganization. Sometimes reorganization is only needed in a subtree, leading to less overhead. On the other hand, if a new level is needed in the tree, then the whole tree needs to be reorganized. Even though this situation is less likely to occur than the other case, it has more overhead involved.

Regardless of their cause the reorganizations require some policy decisions. Choosing the domain and the splitting point in the node (for the related partition) are important aspects and the methods used are analogous to grid file methods. For instance one way of choosing domains is to do so cyclically. Similarly this cyclic method may be modified if something is known about queries. It is desirable to favor some domains if they are expected to be referred to more often. Also given the splitting domain, the splitting point should be chosen so as to have approximately the balanced number of records in the partitions throughout the whole tree. The midpoint or median value can be among the choices. In some cases, as with duplicate records, special care may be needed. Overall, the reorganization of the tree affects the size of the tree. In the worst case, the tree size can double.

From these considerations it becomes evident that the splitting process is a complex one, and its cost can not be expressed in a simple form because of the large number of different situations and possible policy decisions that are made in any implementation. This aspect has been left for future studies.

Insertion Performance: Results

In this chapter an attempt has been made to quantify the cost of insertions for comparing the two structures. The results may be summarized as follows.

a. The cost of an insertion can be computed on a statistical basis, since a splitting operation will be encountered on some insertions while others will only cost as much as a point query.

b. The expected cost can be expressed as a weighted average of the cost of a splitting operation and the cost of a point query, the weights being the probability of splitting per insertion, P(x), and 1 - P(x), respectively.

c. A splitting operation may cost a different amount each time it occurs, but the cost is, in general, much higher than that of an insertion without splitting. This is true for both structures. Although not quantified, it appears reasonable to expect a comparable cost of splitting in the two structures.

d. The probability of splitting per insertion is the inverse of the return period of splitting, measured as the number of records that can be inserted without causing a splitting. The return period depends on the mean bucket occupancy ratio maintained by a file structure, e0, and the occupancy ratio at splitting, e_split (which is a function of b and c).

From this discussion it follows that a comparison of the two structures may be made, roughly, based on the return period of splitting alone. For the same database size, if the two file structures use the same bucket capacity then there will be about the same number of buckets. So e_split will be approximately the same. Then the main

factor that governs the return period is the mean occupancy ratios maintained. Under these conditions the structure that maintains a higher bucket occupancy ratio will create more frequent splittings; therefore its insertion performance will be inferior.

Another conclusion that may be derived from the studies of this chapter is that if insertion performance is an important issue in a given implementation, then one has to develop the structure, with suitable policy decisions, to maintain a compatible bucket occupancy ratio. The numerical results presented here may be used as a guide.

CHAPTER VI

MEMORY UTILIZATION

Introduction

To study the memory utilization of grid file and k-d-B-tree, we need to investigate two separate aspects of memory usage:

a. how efficiently the buckets are utilized by two structures;

b. the storage required by the access mechanism, i.e., how much storage is needed by the grid file and k-d-B-tree structures themselves.

These two aspects of memory usage will be disscussed in the following.

Bucket Utilization

In simulation studies and actual implementations it has been reported that bucket occupancy ratios of 0.70 in the grid file case, and 0.60 in the k-d-B-tree case can be maintained [14, 19]. Improving these occupancy ratios may be possible for both access mechanisms because of the flexibility in the policy decisions that can be made. In general, it is desirable to obtain a high occupancy ratio for efficient utilization of memory resources. However, the bucket occupancy ratio is closely related to the frequency of splitting as discussed in Chapter 5. Briefly, if the bucket occupancy ratio is kept large, then splitting is more frequent. Clearly, frequent splittings degrade the insertion

performance. Therefore one has to consider this interaction between the two in the design of access mechanisms for a specific application. If efficient memory utilization is more important, then one has to sacrifice from insertion speed, or vice versa. We have supplied some data that may be useful in this regard.

For both structures the buckets that hold the actual database records are expected to be kept in the secondary memory. To gain some idea about this memory requirement we may calculate the number of buckets needed in both grid file and k-d-B-tree, using the general relationships derived in Chapter 3. Table VI gives the results obtained in this manner; it shows the number of buckets needed for different database sizes and for different bucket capacities. Bucket occupancy ratios of 0.70 and 0.60 are used in the calculations for grid file and k-d-B-tree respectively. Table VI shows that the number of buckets increases when the database size increases. It is natural to expect that more buckets are needed for larger database if the same bucket capacity is used. Since the k-d-B-tree has a smaller average bucket occupancy ratio, more buckets (17%) are required to accommodate the same database. When the bucket capacity is increased, the number of buckets decreases for both structures.

Size of Access Mechanism

To compare the memory requirements of the two access mechanisms, we need to calculate the total number of bytes requires to store the grid file and k-d-B-tree structures.

TABLE VI

NUMBER OF BUCKETS

N = 1000	c=10	c=30	c=100	c=300	c=1000
grid file	143	48	14	5	1
k-d-B-tree	167	56	17	6	2

N = 10000	c=10	c=30	c=100	c=300	c=1000
grid file	1429	476	143	48	14
k-d-B-tree	1667	556	167	56	17

N = 50000	c=10	c=30	c=100	c=300	c=1000
grid file	7143	2381	714	238	71
k-d-B-tree	8333	2778	833	278	83

N = 100000	c=10	c=30	c=100	c=300	c=1000
grid file	14286	4762	1429	476	143
k-d-B-tree	16667	5556	1667	556	167

N = 500000	c=10	c=30	c=100	c=300	c=1000
grid file	71429	23810	7143	2381	714
k-d-B-tree	83333	27778	8333	2778	833

N = 1000000	c=10	c=30	c=100	c=300	c=1000
grid file	142857	47619	14286	4762	1429
k-d-B-tree	166667	55556	16667	5556	1667

Grid file structure consists of linear scales and the grid directory. Linear scales are one-dimensional arrays containing the key values separating the intervals. Let us denote the number of bytes needed to store one key value by v. This size naturally will depend on the application.

The size of the grid directory is the dominating memory overhead of the grid file structure. The grid directory consists of the grid blocks, and each grid block contains a bucket address (or a pointer). Let us denote the number of bytes needed to hold one such pointer by p. The number of grid blocks in the grid directory is given by n^k as derived in Chapter 3.

Since each linear array is associated with a different key, the number of keys gives the total number of the linear scales needed. Each linear scale contains n key values, and the grid array contains n^k entries. Thus the total size of the grid file access mechanism, s_g, may be calculated as follows.

$$s_g = (kn)\ v + (n^k)\ p \qquad (6.1)$$

The memory requirement of the k-d-B-tree is the number of bytes to store the information at the nodes at each level. This size can be calculated as follows. There are m-1 key values (each of which occupies v bytes) and m pointers (of p bytes each) in each node. The memory requirement, or "size" of k-d-B-tree is then

$$s_k = \text{nodes}\ [(m-1)\ v + m\ p] \qquad (6.2)$$

where nodes is the total number of nodes in the tree;

$$\text{nodes} = 1 + \sum_{i=1}^{h-1} m^i = (m^h-1)\ /\ (m-1) \qquad (6.3)$$

The number of bytes needed to store key values, v, and that of pointers, p, have to be known to be able to evaluate these formulas numerically. However, to compare the memory requirements of the two structures in this study we will simplify these formulas by assuming that the two parameters, v and p, are approximately equal, say

$$v = p = mu \qquad (6.4)$$

This way it is possible to do a comparative evaluation in terms of a common "memory unit" mu.

Having established the memory sizes in this manner, we can easily compute the access mechanism sizes for various database sizes and characteristics. The results of these calculations are given in Tables VII and VIII in terms of memory units.

Comparison

In Table VII the effect of dimension, k, and database size on access mechanism size in terms of "memory units" defined above, is investigated. Table VII also includes the effect of the number of cycles, f, for k-d-B-tree. The bucket capacity, c, is kept constant while other parameters vary in these calculations.

From the results in Table VII we observe how access mechanism size increases with increasing database size for both structures. The rate of increase is almost the same as the rate of increase in database sizes. An interesting observation that can be made is that the grid file size slightly fluctuates around a constant value with increasing k for a given database size. This shows a dynamic capacity of grid file which seems to absorb the potential increase in overhead as dimension increases. But in the case of k-d-B-tree, the memory requirement for

the tree increases with increasing dimension. It also increases with number of cycles, f. But the rate of increase with increasing f is not substantial.

In practically all the cases considered in constructing Table VII, the grid file access mechanism has a smaller memory requirement than k-d-B-tree. Grid file size is almost always smaller than the size of a one-cycle k-d-B-tree. The difference becomes even more significant with the increasing cycles.

In addition to more efficient memory utilization to store the structure, the stable size of grid file makes it a more desirable choice for high dimensions since the k-d-B-tree size increases considerably for large dimensions. For example, a four-cycle k-d-B-tree typically requires about four times the memory that a grid file occupies for k=10.

In the preceding comparative study a constant bucket capacity, c=100, was used. This could have an effect on the results. In fairness to k-d-B-tree we need to look at the structure sizes for other c values. We repeated the same size calculations for c values between 10 and 1000, and obtained the results given in Table VIII. As should be expected, the structure sizes decrease as c increases. Other than this, our conclusions do not change; similar observations can be made in Table VIII for all c values. The results may be summarized as follows:

1. The structure size in grid file does not change appreciably with dimension. The size of a comparable k-d-B-tree, on the other hand, increases with dimension.

2. The size of a k-d-B-tree also increases with increasing number of cycles.

3. In almost all of the cases considered a grid file requires less memory than even a one-cycle k-d-B-tree.

Therefore, we conclude that the grid file is more efficient in its memory utilization.

TABLE VII

COMPARISON OF ACCESS MECHANISM SIZES
(c= 100)

N = 1000

k	grid file	k-d-B-tree f=1	f=2	f=3	f=4
2	39.26	36.42	46.69	57.52	68.51
3	37.74	41.41	57.52	74.02	90.63
4	37.82	46.69	68.51	90.63	112.83
5	38.35	52.07	79.55	107.27	135.05
6	39.06	57.52	90.63	123.94	157.30
7	39.87	63.00	101.72	140.61	179.55
8	40.74	68.51	112.83	157.30	201.81
9	41.63	74.02	123.94	173.99	224.07
10	42.55	79.55	135.05	190.68	246.33

N = 10000

k	grid file	k-d-B-tree f=1	f=2	f=3	f=4
2	319.52	345.24	395.22	454.42	516.33
3	305.47	368.12	454.42	547.74	642.95
4	302.16	395.22	516.33	642.95	771.02
5	301.21	424.30	579.35	738.93	899.67
6	301.11	454.42	642.95	835.30	1028.61
7	301.42	485.18	706.88	931.88	1157.72
8	301.94	516.33	771.02	1028.61	1286.93
9	302.58	547.74	835.30	1125.43	1416.22
10	303.32	579.35	899.67	1222.32	1545.55

N = 50000

k	grid file	k-d-B-tree f=1	f=2	f=3	f=4
2	1504.16	1694.53	1855.01	2067.22	2296.21
3	1462.36	1763.63	2067.22	2413.69	2772.28
4	1453.16	1855.01	2296.21	2772.28	3257.72
5	1449.95	1957.90	2532.39	3135.88	3746.96
6	1448.71	2067.22	2772.28	3502.02	4238.13
7	1448.33	2180.45	3014.32	3869.62	4730.39
8	1448.41	2296.21	3257.72	4238.13	5223.35
9	1448.74	2413.69	3502.02	4607.26	5716.77
10	1449.25	2532.39	3746.96	4976.82	6210.53

TABLE VII (Continued)

N = 100000

k	grid file	k-d-B-tree f=1	f=2	f=3	f=4
2	2964.05	3373.16	3640.40	4013.06	4421.62
3	2899.71	3484.76	4013.06	4632.38	5278.09
4	2886.39	3640.40	4421.62	5278.09	6155.05
5	2881.70	3819.90	4845.81	5934.77	7040.39
6	2879.74	4013.06	5278.09	6597.03	7929.96
7	2878.96	4214.61	5715.08	7262.48	8821.98
8	2878.77	4421.62	6155.05	7929.96	9715.50
9	2878.93	4632.38	6597.03	8598.81	10610.03
10	2879.31	4845.81	7040.39	9268.58	11505.30

N = 1000000

k	grid file	k-d-B-tree f=1	f=2	f=3	f=4
2	28909.48	33461.44	34939.65	37442.20	40360.92
3	28663.15	34010.37	37442.20	41899.55	46683.90
4	28623.43	34939.65	40360.92	46683.90	53267.08
5	28610.35	36114.40	43471.49	51607.91	59958.30
6	28604.62	37442.20	46683.90	56603.59	66704.38
7	28601.75	38868.50	49956.39	61640.82	73482.10
8	28600.28	40360.92	53267.08	66704.38	80279.54
9	28599.58	41899.55	56603.59	71785.42	87090.31
10	28599.34	43471.49	59958.30	76878.98	93910.56

N = 1E+07

k	grid file	k-d-B-tree f=1	f=2	f=3	f=4
2	286783.44	333740.38	342009.31	359298.28	381017.88
3	285911.94	336415.75	359298.28	392776.12	430023.41
4	285806.78	342009.31	381017.88	430023.41	482120.47
5	285776.22	349872.47	404922.25	468934.47	535525.12
6	285763.12	359298.28	430023.41	508712.09	589600.12
7	285756.72	369788.06	455841.94	548995.94	644061.75
8	285752.94	381017.88	482120.47	589600.12	698768.38
9	285750.72	392776.12	508712.09	630420.12	753638.19
10	285749.44	404922.25	535525.12	671391.00	808622.00

TABLE VIII

COMPARISON OF ACCESS MECHANISM SIZES
(N= 100000)

c = 10

k	grid file	k-d-B-tree f = 1	f = 2	f = 3	f = 4
2	28909.48	33461.44	34939.65	37442.20	40360.92
3	28663.15	34010.37	37442.20	41899.55	46683.90
4	28623.43	34939.65	40360.92	46683.90	53267.08
5	28610.35	36114.40	43471.49	51607.91	59958.30
6	28604.62	37442.20	46683.90	56603.59	66704.38
7	28601.75	38868.50	49956.39	61640.82	73482.10
8	28600.28	40360.92	53267.08	66704.38	80279.54
9	28599.58	41899.55	56603.59	71785.42	87090.31
10	28599.34	43471.49	59958.30	76878.98	93910.56

c = 30

k	grid file	k-d-B-tree f = 1	f = 2	f = 3	f = 4
2	9718.99	11184.65	11836.77	12840.35	13974.85
3	9587.41	11441.51	12840.35	14566.36	16391.88
4	9563.33	11836.77	13974.85	16391.88	18887.10
5	9555.06	12314.07	15168.00	18259.30	21414.55
6	9551.44	12840.35	16391.88	20148.11	23958.35
7	9549.73	13397.41	17633.78	22049.32	26511.48
8	9548.96	13974.85	18887.10	23958.35	29070.48
9	9548.72	14566.36	20148.11	25872.52	31633.38
10	9548.81	15168.00	21414.55	27790.38	34199.15

c = 50

k	grid file	k-d-B-tree f = 1	f = 2	f = 3	f = 4
2	5865.47	6723.40	7169.69	7827.80	8561.80
3	5767.92	6903.75	7827.80	8942.71	10114.46
4	5749.06	7169.69	8561.80	10114.46	11711.56
5	5742.50	7484.44	9329.37	11310.00	13326.92
6	5739.66	7827.80	10114.46	12517.70	14951.49
7	5738.37	8188.95	10909.77	13732.39	16581.36
8	5737.88	8561.80	11711.56	14951.49	18214.54
9	5737.82	8942.71	12517.70	16173.52	19850.04
10	5738.04	9329.37	13326.92	17397.62	21486.98

TABLE VIII (Continued)

c = 100

k	grid file	k-d-B-tree			
		f = 1	f = 2	f = 3	f = 4
2	2964.05	3373.16	3640.40	4013.06	4421.62
3	2899.71	3484.76	4013.06	4632.38	5278.09
4	2886.39	3640.40	4421.62	5278.09	6155.05
5	2881.70	3819.90	4845.81	5934.77	7040.39
6	2879.74	4013.06	5278.09	6597.03	7929.96
7	2878.96	4214.61	5715.08	7262.48	8821.98
8	2878.77	4421.62	6155.05	7929.96	9715.50
9	2878.93	4632.38	6597.03	8598.81	10610.03
10	2879.31	4845.81	7040.39	9268.58	11505.30

c = 500

k	grid file	k-d-B-tree			
		f = 1	f = 2	f = 3	f = 4
2	619.24	683.92	766.21	868.16	976.10
3	596.32	720.68	868.16	1031.13	1198.36
4	590.99	766.21	976.10	1198.36	1423.87
5	589.23	816.02	1086.59	1367.33	1650.71
6	588.71	868.16	1198.36	1537.18	1878.20
7	588.76	921.68	1310.88	1707.53	2106.09
8	589.12	976.10	1423.87	1878.20	2334.20
9	589.65	1031.13	1537.18	2049.09	2562.48
10	590.30	1086.59	1650.71	2220.12	2790.87

c = 1000

k	grid file	k-d-B-tree			
		f = 1	f = 2	f = 3	f = 4
2	319.52	345.24	395.22	454.42	516.33
3	305.47	368.12	454.42	547.74	642.95
4	302.16	395.22	516.33	642.95	771.02
5	301.21	424.30	579.35	738.93	899.67
6	301.11	454.42	642.95	835.30	1028.61
7	301.42	485.18	706.88	931.88	1157.72
8	301.94	516.33	771.02	1028.61	1286.93
9	302.58	547.74	835.30	1125.43	1416.22
10	303.32	579.35	899.67	1222.32	1545.55

CHAPTER VII

CONCLUSIONS

Summary and Conclusions

The grid file and k-d-B-tree structures have been compared analytically to assess their relative efficiency in accessing and inserting records, and memory utilization. The results may be summarized as follows.

First a number of useful relationships have been derived for both structures. These relate various quantities such as the number of buckets, bucket size, database size, bucket occupancy ratio, and dimension. These may be used, for example, to estimate the average number of intervals in one linear scale of a grid file, n, and the order, m, in a comparable k-d-B-tree. Such calculations were necessary in the course of this study so that fair comparisons could be done. However, these formulas can also be used for other purposes.

Types of queries possible in a multi-dimensional database have been classified. The average number of block assesses required in a query is taken as the measure of performance, and analytical expressions have been developed to estimate the number of block accesses for each type of query. Using these expressions some parametric, but hypothetical situations have been considered to obtain estimates in order to compare the two file structures.

From the point of view of query performance, it was found that, in general, grid files access records "faster," as measured by the number of block accesses necessary to reach the desired records. One exception to this is a one-cycle k-d-B-tree, which seems slightly more efficient for small (k<8) dimensions in the case of range queries, but its performance slows down rapidly with increased number of cycles.

In both structures, but considerably more so in the grid file, the number of block accesses decreases as the dimension increases. This should be expected because these structures have been designed as multikey access mechanisms. Further, this rate of decrease with k increases with database size, N. That is, the query efficiency of the structures tend to increase with larger databases.

Next the cost of an insertion was considered for comparing the two structures. It has been found that this cost can be computed on a statistical basis, since a splitting operation will be encountered on some insertions while others will only cost as much as a point query. The expected cost can be expressed as a weighted average of the cost of a splitting operation and the cost of a point query. However, because a splitting operation is much costlier than a query, the average return period of splitting may be taken as a measure of comparison.

It was found that the return period depends on the mean bucket occupancy ratio maintained by a file structure, e0, and the occupancy ratio at splitting, e_split, which in turn is a function of b and c. This functional relationship has been determined. This analysis indicates that the structure that maintains a higher bucket occupancy ratio (grid file) will create more frequent splittings; therefore its insertion performance will be inferior.

For comparing the relative efficiency of grid file and k-d-B-tree, two aspects of memory utilization have been isolated for comparing the two structures: (1) bucket utilization, and (2) size of access mechanism. Bucket utilization has not been evaluated analytically in this study. The data available from the literature suggests that the difference between the two structures in the usage of buckets are not large, with k-d-B-tree requiring approximately 17% more buckets for the same database as compared to grid file.

A measure is proposed to compare the memory requirement of the structure itself (i.e., the access mechanism). This may be called the structure size in "memory units" (One memory unit is defined as a group of bytes required to hold a pointer or a key value). It was found that the structure size in a grid file does not change appreciably with dimension. On the other hand, the size of a comparable k-d-B-tree increases with dimension and also with increasing number of cycles. In almost all of the cases considered grid file requires less memory than even a one-cycle k-d-B-tree.

Therefore it is concluded that the grid file is more efficient in its memory utilization.

Suggestions for Future Work

Further analysis of insertion performance may be necessary to determine a more accurate estimation of the cost of an insertion. For this purpose the cost of a splitting has to be calculated in both structures. Deletion performance can be studied in a similar fashion for both structures. It appears that an implementation of k-d-B-tree may be quite complex in building and reorganizing the structure due to a

number of different policy decisions that can be made. Therefore, implementations of both structures especially k-d-B-tree may be useful to study these aspects. Applications with real databases will be helpful to see how the results of the analytical approach apply to a real situation.

BIBLIOGRAPHY

1. Knuth, D. E. The Art of Computer Programming. Vol. 3, Sorting and Searching. Addison-Wesley, Reading, Mass., 1973, 481-492, 471-479.

2. Lum, V. Y. "Multiattribute Retrieval with Combined Indices." Communications of the ACM 13, 11 (1970), 660-665.

3. Mullin, J. K. "Retrieval Update Speed Trade-offs Using Combined Indices." Communications of the ACM 14, 12 (1971), 775-778.

4. Vallarino, O. "On the Use of Bit Maps for Multiple Key Retrieval." ACM SIGPLAN Notices 11 (March 1976), 108-114.

5. Batory, D. S. "On Searching Transposed Files." ACM Transactions on Database Systems 4, 4 (1979), 531-544.

6. Rothnie, J. B., Lazano, T. "Attribute-Based File Organization in a Paged Environment." Communications of the ACM 17, 2 (1974), 63-69.

7. Gopalakrishna, V., Madhavan, C. E. Veni. "Performance Evaluation of Attribute-Based Tree Organization." ACM Transactions on Database Systems 6, 1 (March 1980), 69-87.

8. Finkel, R. A., Bentley, J. L. "Quad Trees: A Data Structure for Retrieval on Composite Keys." Acta Informatica 4, 1 (1974), 1-9.

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

10. Bentley, Jon L. "Multidimensional Search Trees in Database Applications." IEEE Transactions on Software Engineering SE-5, 4 (1979), 333-340.

11. Kashyap, R. L., Subas, S. K. C., Yao, S. Bing. "Analysis of the Multiple-Attribute-Tree Data- Base Organization." IEEE Transactions on Software Engineering SE-3, 6 (1977), 451-466.

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

13. Lee, D. T., Wong, C. K. "Quintary Trees: A File Structure for Multidimensional Database Systems." ACM Transactions on Database Systems 5, 3 (1980), 339-353.

14. Robinson, Jon L. "The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes." Proc. ACM SIGMOD, Ann Arbor (April 1981), 10-18.

15. Sharma, K. D., Rani Rekna. "Choosing Optimal Branching Factors for K-D-B-Trees." Information Systems 10, 1 (1985), 127-134.

16. Ouksel Mohamed, Scheurmann. "Multidimensional B-Trees : Analysis of Dynamic Behavior." BIT 21 (1981), 401-418.

17. Vaisnavi, V. K., Kriegel, H. P., Wood, D. "Optimum Multiway Search Trees." Acta Informatica 14, (1980), 119-133.

18. Liou, J. H., Yao, S. B. "Multidimensional Clustering for Database Organizations." Information Systems 2 (1977), 187-198.

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

20. Merrett, T. H. "Multidimensional Paging for Efficient Database Querying." Proc. ICMOD (Milano, Italy, June 1978), 227-289.

21. Merrett, T. H., Otoo, E. J. "Dynamic Multipaging: A Storage Structure for Large Shared Data Banks." Rep. SOCS-81-26, McGill Univ., 1981.

22. Burkhard, W. A. "Interpolation Based Index Maintenance." Proc. ACM Symp. Principles of Database Systems (1983), 76-89.

APPENDIXES

APPENDIX A

NUMBER OF ACCESSED INTERVALS IN A RANGE QUERY

The derivation of Equations 4.2 and 4.3, giving the expected number of accessed intervals in a range query, is presented here. The relationship will be derived with reference to grid file, but, as explained in the main text, it is equally valid in k-d-B-tree when n's are replaced by m's in the equations.

The problem may be studied as follows:
There are n equal subdivisions in a linear scale, defined by (n-1) delimeters d_i. A range query is considered. It is assumed that the query range can have any length, from zero to the entire range of the linear scale. The question we pose here is "what is the expected number of intervals covered by an arbitrary range query?"

To calculate the expected number of intervals covered by a random range query, we consider the number of different possible situations: a query range can be smaller than the width of one interval, it can be larger than one interval, but less than two intervals, etc. These situations can be studied on the example in Figure 19 where seven intervals are taken. We see that the query range can have a length of one through n-1 intervals. Furthermore, each of such ranges can be situated at a different location with respect to the linear scale. Counting all the possible situations in this example, we find 48. We call this sum S. It may be noted that a point query is included in the

case where the length of the query range is less than one interval, and a full range query is included in the last case (length>6).

If we assume that each of the possible situations are equally likely, then the probability of one is 1/S. In Figure 19 the number of intervals covered in each of these S situations, determined by inspection, is also shown. At the bottom of the figure, the number of one, two, ..., seven intervals covered is totalled. For example, seven of the 48 cases will cover one interval, 12 will cover 2 intervals, etc. Thus, the probability of covering one interval is P(1)=7/48, that of two intervals is P(2)=12/48, etc. Then it is a straightforward calculation to determine the expected number of intervals covered:

$$E(i) = (1)\ P(1) + (2)\ P(2) + \ldots + (7)\ P(7) \tag{A.1}$$

or, in general,

$$E(i) = \sum_{i=1}^{n} i\ P(i) \tag{A.2}$$

where

$$P(i) = \begin{cases} n/S, & i=1 \\ 2(n-i+1)/S, & i=2,3,4,\ldots,n-1 \\ 1/S, & i=n \end{cases} \tag{A.3}$$

and

$$S = 1 + 2\sum_{i=2}^{n-1} (n-i+1) + n = n^2 + 6n - 9 \tag{A.4}$$

Since the expected number of covered intervals, E(i), is used in further derivations in the main text, we call it n_a, short for "number of accessed intervals." Evaluating Eqs. (A.1) through (A.4) for n=3-30,

the relationship shown in Figure 20 is obtained. For use in the calculations, as in Chapter 4, it is desirable to have a simple expression for this relationship. Therefore the log-log linear relationship,

$$n_a = a\ n^b \tag{A.5}$$

has been fit, where b is the slope of the line and a is the value of the function at n=1. For accurate fitting, it has been found necessary to divide the n range in two at n=9. Thus,

$$n_a = 0.80\ n^{0.71}, \quad n < 9 \tag{4.2}$$

$$n_a = 0.57\ n^{0.87}, \quad n >= 9 \tag{4.3}$$

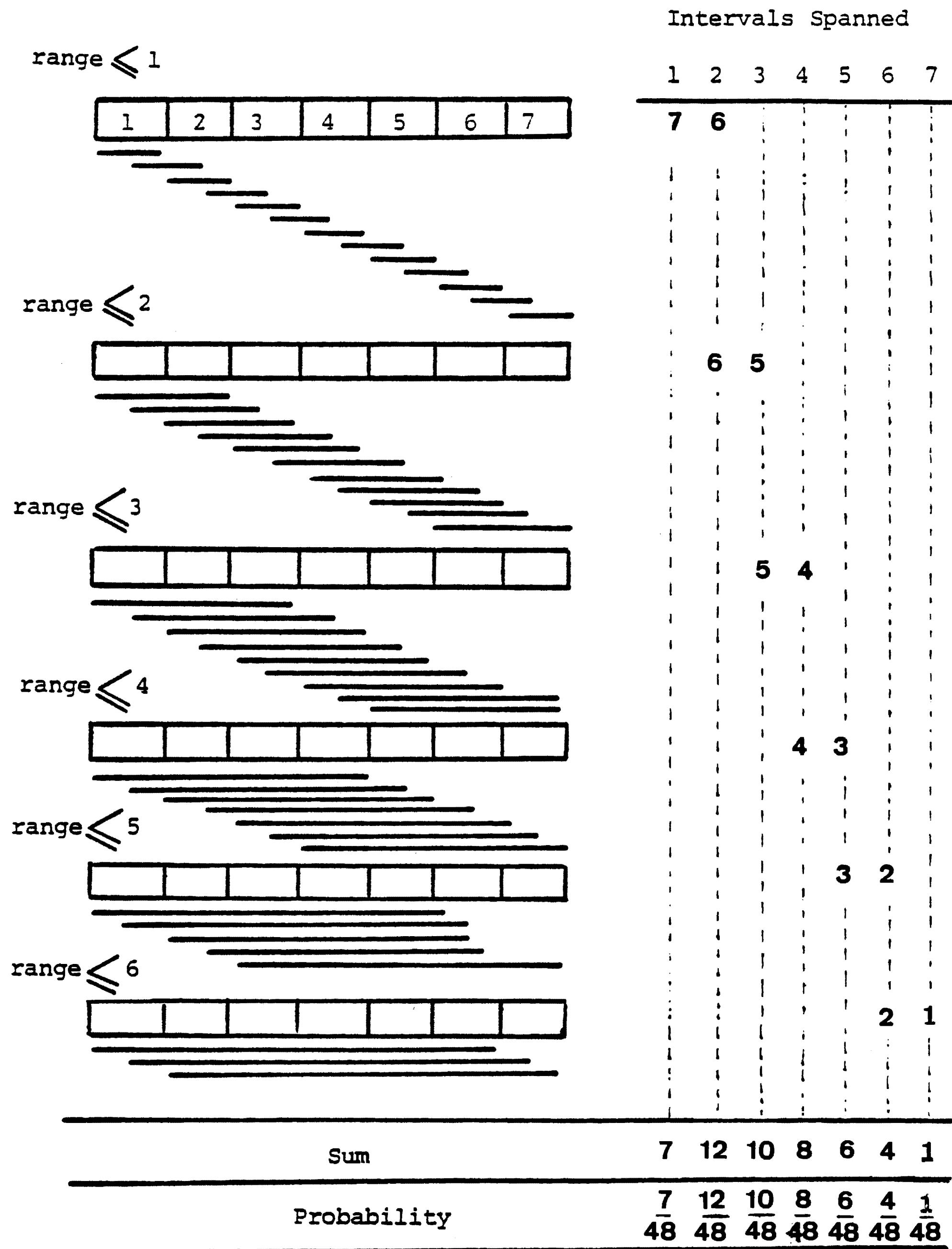


Figure 19. Number of Intervals Spanned in a Range Query (n=7)

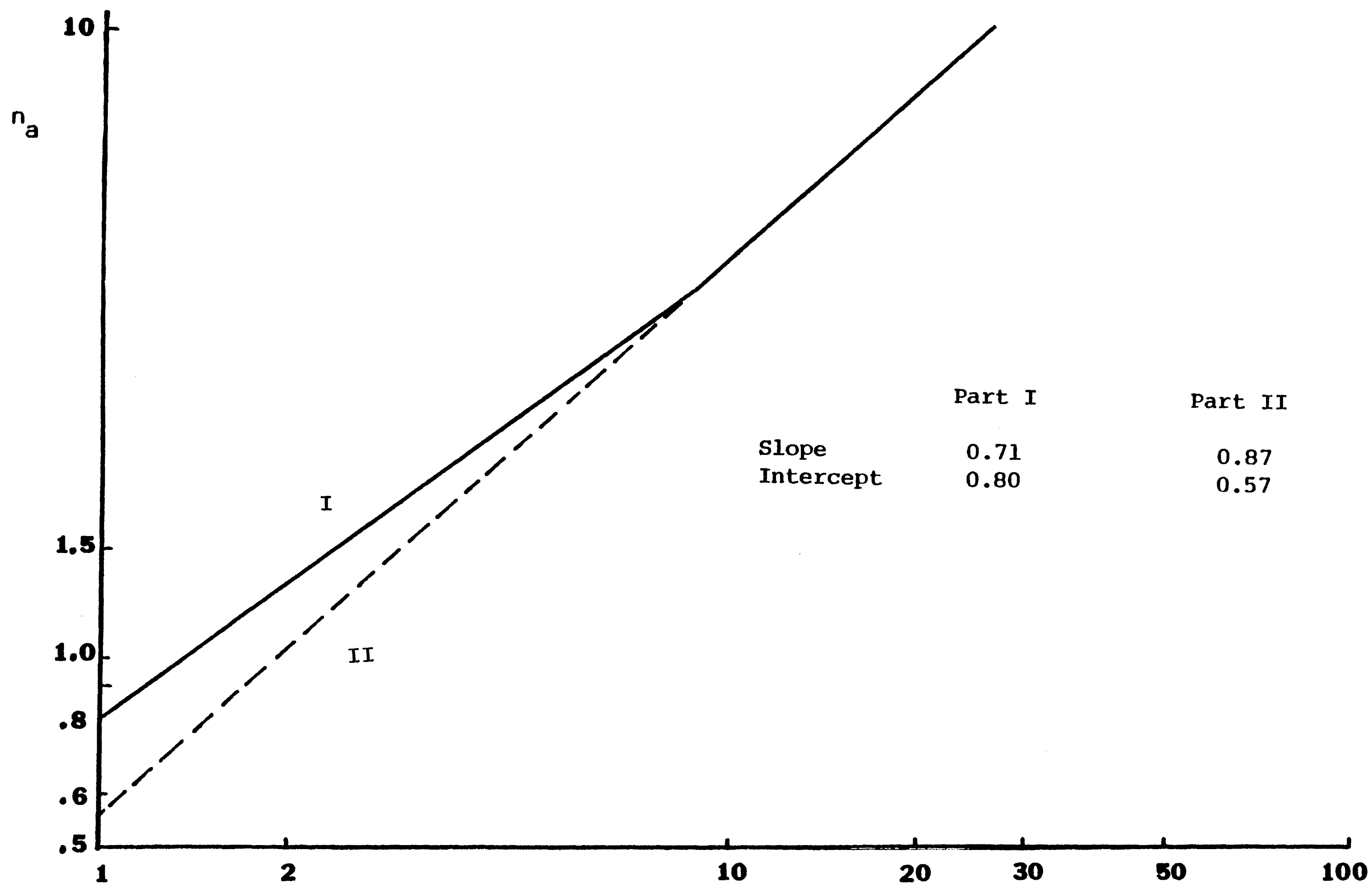


Figure 20. Expected Number of Intervals in a Range Query

APPENDIX B

BUCKET OCCUPANCY AT SPLITTING

We consider a hypothetical set of b buckets, each of which can hold c records, and consider records being inserted. This will be simulated by generating random numbers. Since we assume that the database is uniform, the probability of a new record to "hit" a given bucket is 1/b. Now if we assume a key value range of 0 to 1, then the first bucket will take records in the range (0, 1/b), the next between (1/b, 2/b), and so on, until bucket number b which will take records in the range ((b-1)/b, 1). The total available capacity will be "bc", and at maturity there will be

$$N = b\ c\ e \tag{B.1}$$

records where e is the average occupancy ratio.

To study how the buckets are filled, we generate uniformly distributed random numbers between 0 and 1, "put" each record in its bucket according to the value of the random number, and continue this until one bucket overflows. This way we will see both the average occupancy ratio, e, and its distribution over the buckets at the moment splitting occurs. This value of e is called e_split in Chapter 5.

A "b" range of 3 to 300, and a "c" range of 3 to 100 have been chosen. It was found that the scatter of the results were larger for smaller combinations of b and c, but much less for larger values. On the other hand, computation time was greater for the large b and c.

Therefore the number of repetitions for each case varied. To obtain reliable results using reasonable numbers of repetitions the running average was printed at the end of each trial for a given case, and as this running average stopped fluctuating the test was stopped. At least three, but in general four decimal digits were obtained accurately for each case. The results are presented in Table IX.

TABLE IX

RESULTS OF NUMERIC EXPERIMENTS

number of buckets (b)	bucket capacity (c)	number of trials	e_split mean	e_split st. dev.
3	3	200	.690	.176
3	10	100	.804	.118
3	30	100	.867	.077
3	100	100	.917	.044
10	3	100	.453	.147
10	10	100	.618	.098
10	30	100	.769	.064
10	100	50	.848	.045
10	1000	10	.960	.015
30	3	200	.330	.112
30	10	100	.533	.085
30	30	75	.724	.056
30	100	20	.809	.038
100	3	50	.252	.095
100	10	50	.448	.067
100	100	50	.782	.028
100	1000	10	.925	.012

APPENDIX C

INSERTIONS WITH SPLITTING

The dynamic behavior of the grid file and the k-d-B-tree is best explained by tracing an example: that is, building up the structures under repeated insertions. In order to simplify the description, the following small two-dimensional database is used for both structures:

Rec No.	1	2	3	4	5	6	7	8	9	10
Name	L	K	T	T	I	G	N	D	K	T
Age	20	10	80	40	90	45	45	70	30	20

Figures 21 and 22 show the grid file and the k-d-B-tree during the insertions respectively. Bucket capacity (c) is assumed to be 3 in both cases.

In Figure 21, instead of showing the grid directory, whose elements are in one-to-one correspondence with the grid blocks, we draw the bucket pointers as originating directly from the grid blocks. Each "dot" in the search space represents a record.

Initially, a single bucket (bucket 1) is assigned to the entire record space. First three records are inserted without causing any problem (Part A). When record 4 comes, it causes bucket overflow, the record space is split, a new bucket (bucket 2) is made available. Midpoint value is chosen as the splitting point. Those records that lie

in one half of the space are moved from the old bucket to the new one. When bucket 1 overflows (because of record 6) again, its grid block is split according to some splitting policy: we assume the simplest splitting policy of alternating directions. After splitting, records of bucket 1 which lie on the lower left grid block of the search space are moved to a new bucket (bucket 3). Notice that, as bucket 2 did not overflow, it is left alone; its region now consists of two grid blocks (Part B). After the record 8 is inserted, record 9 causes an overflow at bucket 3. This triggers a further refinement of the grid partition and splitting bucket 3 into buckets 3 and 4. Record 10 is inserted without any problem (Part D).

In k-d-B-tree case (Figure 22), node order (m) is assumed to be 3 and the organization of the tree is based on the key "Age" at the beginning level.

The first three records are inserted without causing any problem (Part A). When record 4 comes, it causes bucket overflow. Simply, splitting the bucket and reorganizing the node solves the problem (Part B). After records 5 and 6 are inserted in the newly created bucket, this time record 7 causes bucket overflow. Part C shows the situation after the bucket splitting. After record 8 is inserted, record 9 causes overflow in the first bucket. Bucket splitting becomes necessary. Since the root node is also full, a new level needs to be introduced, requiring the complete reorganization of the tree. New level of the tree is partitioned by using the key "Name". Part D shows the tree after splitting and reorganization. Also record 10 is inserted without any complication. In this example, splitting point is chosen as the median of the key values.

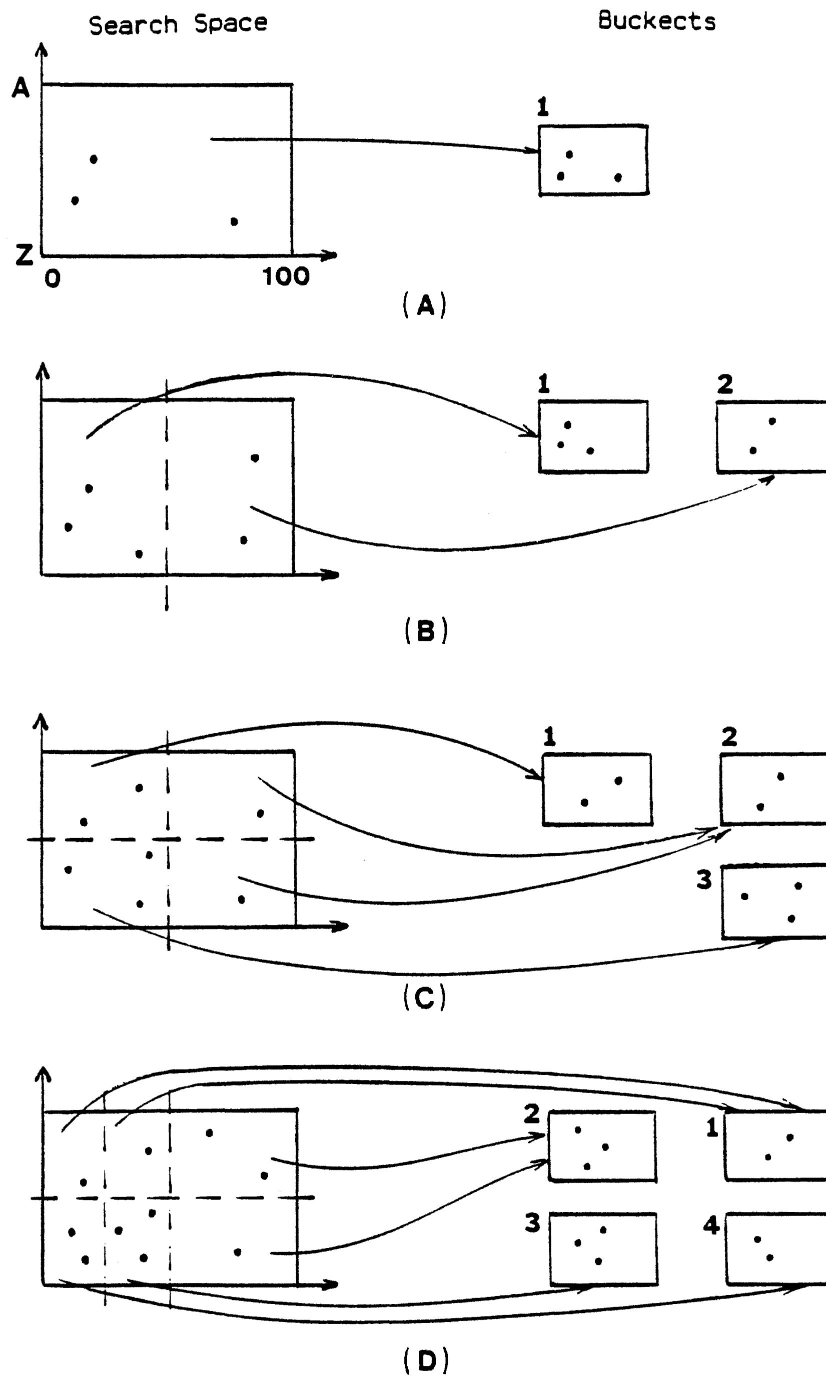
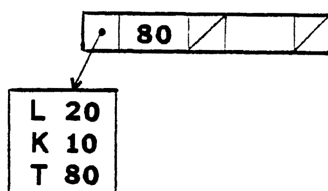
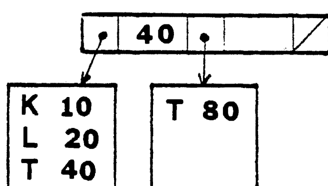


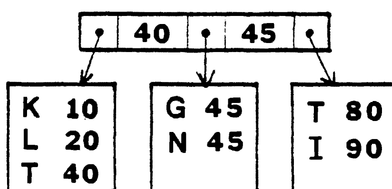
Figure 21. Insertions with Splitting (Grid File)



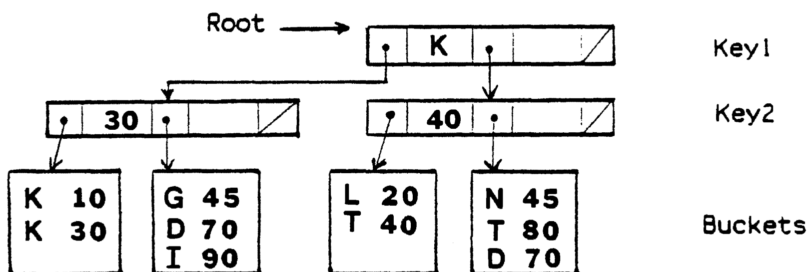
(A)



(B)



(C)



(D)

Figure 22. Insertions with Splitting (K-d-B-tree)

APPENDIX D

TABLE OF SYMBOLS

The following table gives the summary of all the symbols used throughout the thesis in an alphabetical order.

TABLE X

TABLE OF SYMBOLS

Symbol	Description
a_b	Number of buckets accessed
a_g	Number of grid blocks accessed in grid file
a_l	Number of linear scales accessed in grid file
a_n	Number of nodes accessed in k-d-B-tree
b	Number of buckets
c	Bucket capacity (in terms of records)
C1	Cost of an insertion with splitting
C2	Cost of an insertion without splitting
C_b	Cost of a bucket access
C_g	Cost of a grid block access in grid file
C_l	Cost of a linear scale access in grid file
C_n	Cost of a node access in k-d-B-tree
cycle	Frequency of all the keys used in k-d-B-tree

TABLE X (Continued)

Symbol	Description
d_i	Number of delimeters in a linear scale for key i
e	Average bucket occupancy ratio
e_g	Average bucket occupancy ratio in grid file (0.70)
e_k	Average bucket occupancy ratio in k-d-B-tree (0.60)
e0	Mean bucket occupancy ratio
e_split	Average bucket occupancy ratio at the splitting
E_g(query)	Expected cost of a query for grid file
E(i)	Expected number of intervals in a linear scale or in a node for a range query
E(insertion)	Expected cost of an insertion
E_k(query)	Expected cost of a query for k-d-B-tree
f	Usage frequency of the keys in k-d-B-tree
FF	Fully specified Full range query
FP	Fully specified Point query
FR	Fully specified Range query
G	Grid array size (in terms of grid blocks)
h	Tree height (of k-d-B-tree)
k	Dimension (number of keys used)
k_s	Number of keys specified in a range query
key_i	Key value for dimension i
l_i	Lower bound of the domain for the key i
m	Node order in k-d-B-tree

TABLE X (Continued)

Symbol	Description
m_a	Number of pointers covered in a node for a range query in k-d-B-tree
mu	Memory unit (in terms of bytes)
n	Number of intervals in a linear scale in grid file
n_a	Number of intervals covered in a linear scale for a range query in grid file
n_i	Number of intervals in linear scale i in grid file
nodes	Number of nodes in k-d-B-tree
N	Database size (total number of records)
r	Number of grid blocks per bucket
p	Number of bytes needed to store a pointer value
PF	Partially specified Full range query
PP	Partially specified Point query
PR	Partially specified Range query
P(i)	Probability of covering i intervals in a linear scale or in a node for a range query
P(x)	Probability of an occurrence of splitting case
1-P(x)	Probability of non-occurrence of splitting case
S	Sum of intervals covered in a linear scale for a range query
S_g	Size of grid file access mechanism
S_k	Size of k-d-B-tree access mechanism
sigma	Standard deviation for e_split

TABLE X (Continued)

Symbol	Description
T_R	Return period of splitting
u_i	Upper bound of the domain for the key i
v	Number of bytes needed to store a key value

VITA

Hatice Nilufer Anlar Saritepe

Candidate for the Degree of

Master of Science

Thesis: AN ANALYTICAL COMPARISON OF GRID FILE AND K-D-B-TREE STRUCTURES

Major Field: Computing and Information Science

Biographical:

Personal Data: Born in Istanbul, Turkey, October 23, 1958, the daughter of Nakip and Nuran Anlar.

Education: Graduated from Bakirkoy High School, Istanbul, Turkey, in May, 1976; received Bachelor of Science degree in Electrical Engineering from Bogazici University in July, 1981; completed the requirements for the Master of Science degree at Oklahoma State University in December, 1987.

Professional Experience: Programmer, Agricultural Economics Department, Oklahoma State University, April, 1985 to May, 1987; Programmer, Occupational and Adult Education Department, Oklahoma State University, May, 1987 to October, 1987; Research Assistant, Computing and Information Sciences Department, Oklahoma State University, October, 1987 to December, 1987.