12-1986

A General Design Tool for Computer Directories

Edward J. Peeler

A GENERAL DESIGN TOOL FOR COMPUTER DIRECTORIES

by

Edward J. Peeler

A Thesis
Submitted to the
Faculty of The Graduate College
in partial fufillment of the
requirements for the
Degree of Master of Science
Department of Computer Science

Western Michigan University
Kalamazoo, Michigan
December 1986

A GENERAL DESIGN TOOL FOR COMPUTER DIRECTORIES

Edward J. Peeler, M.S.

Western Michigan University, 1986

The primary objective of a directory is to organize information for efficient retrieval. There are many techniques that can be applied to the design of a directory. One particularly useful technique employs the use of inverted files on range attributes. The technique provides an effective directory for a variety of applications and for very large databases. This paper examines the technique and describes the implementation of a general design tool based on these principles.

Peeler, Edward Jay

A GENERAL DESIGN TOOL FOR COMPUTER DIRECTORIES

Western Michigan University M.S. 1986

University
Microfilms
International 300 N. Zeeb Road, Ann Arbor, MI 48106

TABLE OF CONTENTS

TABLE OF CONTENTS--CONTINUED

CHAPTER

APPENDICES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

A significant amount of computer time is engaged in retrieving information from databases. It is important that this task be carried out as efficiently as possible. There are a number of approaches to achieving this goal. Some approaches concentrate on performance increases in the hardware components of a computer system. Others concentrate on devising structures that accomodate efficient retrieval and effective update.

This paper describes a class of structures that provide an effective means for retrieval and update. Algorithms are developed and characterized that provide the file designer with an environment allowing the creation of directories that can accomodate a broad range of requirements.

Fundamental Concepts

Information can be logically divided into components. The most fundamental component is the attribute. An attribute, or field, is an atomic piece of information. It is atomic in the sense that it cannot be divided any further without losing meaning. Examples of attributes include first name, last name, city, state, social security number, etc.

Attributes possess characteristics. Characteristics are used to describe the information being represented by the attribute. Characteristics of attributes include data type, maximum number of characters, data domain, and integrity rules.

Data type describes what kind of information is being represented. Some examples of data type include integer, real number, monetary value, etc. The data domain of an attribute refers to the range of legal values that an attribute value can assume. Integrity rules for an attribute describe what checks are made on the value to determine whether or not an instance of the attribute value is possible.

It usually requires several attributes to describe a real world object. For example, to describe a student, it might be necessary to use the attributes last name, first name, street address, city, state, zip, age, date of birth, etc. When attributes are grouped together, a structure called a record is created. Each attribute value in a record is related to other attribute values in the sense that their combined values describe a single real world object.

It is often desirable to store related records together on the basis of some common characteristic. For example, it might be advantageous to store student records together by year of admittance. Storing records together creates a structure called a file. A file has several characteristics. It typically has a file name, which is used to differentiate it from other files, a location on secondary storage. For example, suppose we wished to describe the real world object employees by storing their employee number , salary, and the department number of the department they were associated with.

Employee ID	Salary	Department
85-100	20000	101
85-110	16000	105
85-120	16000	101
85-130	16000	104
85-140	16000	105
85-150	15000	102
85-160	21000	107
85-170	22000	109
85-180	21000	104
85-190	40000	107
85-200	45000	102
85-210	50000	109
85-220	16000	106
85-230	15000	103
85-240	16000	103
85-250	16000	106

Figure 1. Sample File

A database is a collection of related files. A database can contain the descriptions of many real world objects and their relationships to one another. Each object and relationship may be represented by one or more files. The retrieval of information from a database consists of retrieving records from one or more files. This process requires the use of a mechanism that describes where the desired information resides on some storage device and a procedure for the presentation of the information.

Each record in a file can have an associated record number. This number describes a record's relative position in a file. Thus, retrieval typically involves retrieving lists of addresses corresponding to records in files that satisfy some request for information.

A request for information consists of specifying a predicate or statement about the desired records. There are four basic types of requests or queries (Horowitz, Sahni, 1976):

> (1) Simple: the value of a single attribute is specified.
>
> (2) Range: a range of values for a single attribute is specified.
>
> (3) Functional: some function of the attribute values is specified such as average or median.
>
> (4) Boolean: a boolean combination of the first three query types using logical operators. [p. 479]

There are a number of strategies that can be employed to satisfy queries. One of the simplest strategies is to examine each record in the appropriate files to determine whether or not it satisfies the query, i.e., does the record possess the desired attribute values. While this strategy is simple and easy to implement, it is not efficient. For example, to search a file with 2000000 records with a computer that could access and examine 1000 records per second, it would take approximately 33 minutes to satisfy the simplest query.

Another strategy would be to organize the file and take advantage of the organization when searching for information. One possible organizational strategy would be to linearly order the records of the file based on the value of a particular attribute or group of attributes. This would result in a significant reduction of the number of records that would have to be examined. On the average, the time required to honor a query would be proportional to log n, where n is the number of records in the file and log is base 2.

While this strategy provides a marked improvement over an unorganized file, it does harbor some drawbacks. The most notable drawback involves updating the file. If new records are added to the file or if any of the attribute values used to control the organization change, the file must be reorganized. The reorganization involves reordering the file which requires time proportional to n log n for most files. If the file were very large, the amount of time required to reorganize the file would extend into hours.

Another strategy involves identifying attributes that are important to the retrieval process. Once these fields have been identified for a file, auxiliary files are created based on the values of these fields . These files are called indices. In most files and databases, there will be several attributes that will aid the retreival process. A collection of indices based on these fields is called a directory (Horowitz, Sahni, 1976).

An index may be dense or nondense. A dense index has an entry for every distinct value of the attribute in the source file. If an index is nondense, only certain attribute values are represented in the index. In both cases, an index is generally a collection of pairs of the form: (Field value,Address). The ordered pairs are stored in a separate file with their own organization.

An effective index can be characterized by its behavior. An effective index will provide rapid access to information. It will also provide a convenient and relatively efficient method for maintaining the index should new records be added or existing field values change.

One important component influencing the effectiveness of an index is its structure. The choice of an appropriate structure depends upon many diverse factors. Some important factors include retrieval patterns, frequency of update, and distribution of field values.

If the retrieval patterns are known, structures can be employed that optimize access to frequently specified values. If it is known that only a few values are specified in queries, an index containing only those values used in the retrieval process might prove more effective.

Frequency of update of indexed field values is important when striking a balance between efficiency of retrieval and efficiency of update. Often, a structure optimizing retrieval is difficult to maintain in a dynamic database. If an indexed field value never changes once it has been assigned to a record, it would be proper to employ a structure optimizing retrieval. However, most databases are dynamic and retrieval efficiency must be balanced with update efficiency.

Distribution of indexed field values is important. If each record possesses a unique indexed field value, a structure including field value and address would be appropriate. If an indexed field value is common to several records, a framework providing separate structures for indexed field values and addresses would be more appropriate.

There are many factors that determine the best structure to employ for an index. Unfortunately, there is no one best general structure for an index that accomodates all the factors that influence the structure of an index. However, there is a class of structures based on inverted files that does provide file designers with tools that can take advantage of special a priori knowledge about retrieval patterns as well as when very little is known about retrieval patterns. This paper concentrates on these structures and develops techniques that will in effect create a general directory.

CHAPTER II

INVERTED FILES

Inverted files have found widespread use in databases and file systems for many years. The literature review presented here is a general survey of the applications and modifications that have been applied to inverted files.

Review of Literature

Descriptions of inverted files can be found in most textbooks concentrating on data structures and algorithms. Horowitz and Sahni (1976) provide a fundamental description of the technique and Knuth (1973) describes their efficiency in the processing of boolean queries. Efficient use of inverted files in query optimization has been shown by Lie (1976), Putkanen (198Ø), and Schkolnick (1978).

Many recent publications describe systems utilizing inverted files. Tuttle, Sheretz, Bloise, and Nelson (1983) use inverted files in an interactive diagnositic program called RECONSIDER. Patient findings and their synonyms are matched against inverted files of terms from disease descriptions. The number of matching terms determines a disease score. Sorted scores are used to form a differential diagnosis. Gersting, Conneally, Rogers, and Blum (1982) use inverted files for efficient retrieval from a human pedigree database.

Harding and Willet (198Ø) show how inverted files provide an efficient tool for automatic document classification. Schultheisz, Walker, and Kanaan (1981) use inverted files in a chemical dictionary. Schulthiesz (1981) uses inverted files to retrieve data from TOXLINE, a bibliographic and toxicology composed of 11 different files from different sources. He found inverted files to be an effective tool in handling data from differently structured source files with many replications of bibliographic records. Conrad, Bloom, Cooper, Cannon, Friedman, Horowitz, Krikorian, and Lopez (198Ø) utilized inverted files in a statistical package with a cancer database at Boston University Hospital. This includes only a small sample of the application areas for inverted files.

A number of modifications to inverted files have been developed. Most modifications address the issues of excessive space requirements and access times. Compression techniques which improve space utilization have been developed by Schulthiesz (1981) Jakobsson (198Ø,1982) and by Jakobsson and Nevalainen (198Ø). Combinations of clustering records together with compression techniques have been suggested by Navalainen, Jakobsson, and Berg (1978). This techniuqe improves space utilization and also reduces access time. Motzkin (1979) incorporated inverted files into Normal Multiplication Table directories. In Normal Multiplication Tables, the attribute values and address lists are organized in clusters. Several attributes can be stored in the same table. This technique provides rapid access to single as well as multiple attributes while exhibiting economy of space. Hoffer (1980) concentrates on the process of selecting attributes to be inverted. Cardenas (1975) provides ways to measure the performance of inverted files and suggests that attribute values may be kept in a separate, hierarchical structure with pointers to the address lists. Johnson and Webster (1982) propose an efficient way to update an inverted file. Federowics (1982) developed techniques to model term dispersion in inverted files. This model has been applied to the National Library of Medicine's MEDLINE system. Additional references, especailly regarding early development, can be found in the extensive bibliography compiled by Schkolnick (1978). Uniform organization of inverted files has been proposed by Motzkin, Williams, and Chang (1984).

Description of Inverted File

An index is generally a collection of pairs of the form (Attribute value, Address) where the address corresponds to the relative address of the record in the source file possessing the companion attribute value. Additionally, we assume that all attribute values are distinct in the index. In the event that several records possess the same attribute value, the address component of the index pair is a pointer to another address where a list of addresses of all records possessing the attribute value is maintained. It may also be desirable to store the number of records possessing the attribute value in the index.

Value File		
Value	Count	Address
15000	2	1
16000	7	3
20000	1	10
21000	2	11
22000	1	13
40000	1	14
45000	1	15
50000	1	16

Address File
Address
6
14
2
3
4
5
13
15
16
1
7
9
8
10
11
12

Figure 2. Inverted File on Employee Salary From File in Figure 1.

In this fashion, two files have been created. One file, the value file, contains the attribute value, a count of records possessing the field value, and the beginning address in the address file of the list of addresses corresponding to the records possessing the indicated attribute value. The second file, the address file, is responsible for maintaining the lists of addresses. Note that the addresses corresponding to the same field value are clustered together in the address file.

Retrieving Information With Inverted Files

Retrieving information with inverted files involves accessing three files. (Note that all references to retrievals refer to simple queries.) First, the value file must be accessed to determine whether or not the desired attribute value exists in any of the records in the source file. If the attribute value exists, the system extracts the count of records possessing this value and the pointer to the address file. Next, the system accesses the address file. The pointer extracted from the value file indicates the beginning position of the list of addresses of records possessing the desired attribute value. The system then gathers the list of addresses from the address file. Once the addresses have been gathered, the system can then access the indicated records from the source file.

If the retrieval involves a boolean query, the system can determine the records that satisfy the query without having to examine the records in the source file. (This assumes that the boolean query is based on indexed fields.) The manipulations to satisfy the query can be performed on the record addresses gathered from the address files. If the query involves the OR operator, the system would merely perform a union of the corresponding address lists and eliminate any duplicates. If the query involves the AND operator, the system would perform an intersection of the corresponding address lists.

Updating Inverted Files

Updating an inverted file can be a complex operation. There are two basic update operations; add and delete. Note that a change is a combination of the add and delete operations. The addition of values to an inverted file often introduces another file called a differential file. The purpose of the differential file is to house the addresses of new records. At some specific point in time, the information contained in the differential file is incorporated into the address file. Typically, this is done when noticeable performance degradation occurs in retrieval operations.

There are two possible organizations for the differential file. One organization utilizes a linked list. This organization is suitable for environments where the amount of storage space is limited. The other organization is an unorganized pile file. This organization results in processing savings when the differential file is incorporated into the standard address file. However, it does requires more space and can cause performance degradation when honoring queries.

The introduction of a differential file requires a modification of the value file. A pointer to the differential file must be included, if the linked list version is used, or a flag indicating values reside in the differential file for the unorganized version It is also desirable to store the count of entries in the differential file for determining when the differential file should be incorporated into the address file.

The linked list form of the differential file is a pair of the form (Address, Pointer to next address) where Address corresponds to the address of the record possessing the value and the Pointer to next address is the address of the next record in the differential file possessing the same value. The last entry in the linked list for a given value is a Ø in the pointer to next address. Additionally, the first record of the differential file houses the address of the first available record for the addition of new information to the differential file.

For example, suppose that three records are added to the employee file. Two of the records have a salary field of 16ØØØ and one record has a salary field of 21ØØØ. Further, assume that the record numbers corresponding to the records with 16ØØØ as their salary field are 17 and 18. The record number associated with the salary field of 21ØØØ is 19. Figure 3 provides an illustration of the resulting value file and differential file.

Value File

Value	Count	Address	Diff Ptr
15ØØØ	2	1	Ø
16ØØØ	7	3	2
2ØØØØ	1	1Ø	Ø
21ØØØ	2	11	4
22ØØØ	1	13	Ø
4ØØØØ	1	14	Ø
45ØØØ	1	15	Ø
5ØØØØ	1	1	Ø

Address File

Address
6
14
2
3
4
5
13
15
16
1
7
9
8
1Ø
11
12

Differential File

Source Record Number	Next
Ø	5
17	3
18	Ø
19	Ø

Figure 3. Linked List Differential File

The unorganized differential file consists of records of the form (Value, Address of Source Record). This form consumes more space and requires more time to search since each record in the file must be examined. However, the incorporation of the differential file into the standard address file can be accomplished in much less time since the values are present with the addresses.

Value File

Value	Count	Address	Diff Flag
15000	2	1	no
16000	7	3	yes
20000	1	10	no
21000	2	11	yes
22000	1	13	no
40000	1	14	no
45000	1	15	no
50000	1	16	no

Differential File

Value	Source Record Number
16000	17
16000	18
21000	19

Address File

Address
6
14
2
3
4
5
13
15
16
1
7
9
8
10
11
12

Figure 4. Pile Differential File

When adding new entries to the inverted file, there are two scenarios that can develop. The first is encountered when the attribute value being added to the structure exist in the value file. In this case, the address corresponding to the record being added must be placed in the differential file. While it is possible to add the address to the address file, the amount of processing required to accomplish this is expensive.

The second is encountered when the value being added does not have an entry in the inverted file. In this case, a new value entry must be created and inserted in the value file. It is important that the insertion of the new value preserve the organization of the value file.

There are two situations that develop when adding information to the linked list differential file. The first is encountered when the value does not have any addresses in the differential file, i.e., all corresponding addresses reside in the address file. In this situation, the system must first access the fisrt record of the differential file to determine where the first free record is located in the differential file. This address is placed in the value files's pointer to the differential file. The first record of the differential file is updated to point to the next free record in the differential file. The address of the record possessing the value is placed in the differential file and its pointer to next address is set to zero.For example, suppose we now add a fourth record to our file, with a salary field of 2ØØØØ, and a record number of 2Ø. Figure 5 illustrates the resulting differential file.

Value File

Value	Count	Address	Diff Pt
15ØØØ	2	1	Ø
16ØØØ	7	3	2
2ØØØØ	1	1Ø	5
21ØØØ	2	11	4
22ØØØ	1	13	Ø
4ØØØØ	1	14	Ø
45ØØØ	1	15	Ø
5ØØØØ	1	16	Ø

Address File

Address
6
14
2
3
4
5
13
15
16
1
7
9
8
1Ø
11
12

Differential File

Source Record Number	Next
Ø	6
17	3
18	Ø
19	Ø
2Ø	Ø

Figure 5. New Entry Update of Differential File

The second situation arises when the value has address entries in the differential file. Again, the first record of the differential file must be accessed to determine where the first available record in the differential file is located. This record is updated to point to the next available record. The first available record will become the new differential pointer in the value file. The address is then placed in this record and its pointer to next address is set to address that was housed in the value file's differential pointer. Figure 6 illustrates the resulting files if we were to add a record with a salary of 16ØØØ.

Value File

Value	Count	Address	Diff Ptr
15ØØØ	2	1	Ø
16ØØØ	7	3	5
2ØØØØ	1	1Ø	Ø
21ØØØ	2	11	4
22ØØØ	1	13	Ø
4ØØØØ	1	14	Ø
45ØØØ	1	15	Ø
5ØØØØ	1	16	Ø

Address File

Address
6
14
2
3
4
5
13
15
16
1
7
9
8
1Ø
11
12

Differential File

Source Record Number	Next
Ø	6
17	3
18	Ø
19	Ø
2Ø	2

Figure 6. Existing Entry Update of Differential File

Adding information to the differential file when using the unorganized version is very simple. The system must first determine if the value already has an entry in the differential file. If it does not, the system must set the differential flag and add one to the count of the entries in the differential file. Next, the value and the address of the source record are placed in the free position indicated by the first record in the differential file. The first record of the differential file is then updated to reflect the next free position in the file.

Deleting information from an inverted file involves locating the address of the record being deleted from the source file in the inverted structure. Once the address is located, it is replaced with some out of range value. It should be noted that the system may have to search both the address and differential files for the address. It may be desirable to track the number of deletions. When the number of deletions reaches a threshold value, the entire inverted file could be reorganized, adding new entries from the differential file and removing deleted addresses from the address list.

CHAPTER III

VARIATIONS ON INVERTED FILES

A number of modifications to inverted files have been suggested. Chapter II discussed some of those modifications in the survey of the literature. For example, a balanced tree structure could be utilized as a structure for the value file. The advantage of this organization would be to decrease the amount of time required to search the value file. Another technique for organizing the value file would be hashing. Hashing involves the use of a function called a a hashing function. The hashing function is designed to provide unique or nearly unique addresses for given values. The addresses produced by the hashing function would correspond to the records of the value file.

Another modification is the introduction of range attributes for the values in the value file. This organization would not include a separate value entry for every value that existed in the source file, but would consist of ranges of values. This organization is particularly useful if the domain of possible attribute values is large and the retrieval patterns of the attribute typically involves range queries.

A range can be defined as follows:

Given:

n elements drawn from a domain D.
Domain D is a partially ordered set.
ni represents the ith element from domain D.

A range is a subset of the domain such that
[n1,n2,...,ni] such that i <= j and n1 <= ni.

Further, given two successive ranges:
[n1,n2,...,nk-1] [nk,nk+1,...,nj] then
nk-1 <= nk <= nk+1.

Partition domain D into R ranges
Ø < R <= n.

The boundaries of range r, are the endpoints of the
range, i.e., n1,nk-1.

A restriction on successive ranges can be imposed
Given the ranges define above, a restriction is
defined as:
nk-1 < nk <= nk+1.

For example, given the following domain:
(1,2,2,3,4,5,6,7,7,8,9,1Ø)
we might choose to create the following ranges:
(1,2,2,3) (4,5,6) (7,7,8,9,1Ø).

The boundaries of the above ranges are:
(1,3) (4,6) (7,1Ø).

Further note that the ranges are restricted.

The representation of range attributes in inverted files depends upon the organization of the differential file. When the differential file utilizes a linked list, the inverted file consists of value file of the form (Low range value, High range value, Count, Pointer to address file, Differential count, Pointer to differential file). The organization of the address file does not change. It should be noted that the operations on this form of an inverted file may retrieve records that do not satisfy the query, i.e., superfluous records may be retreived. The rest of this paper examines in detail this variation of inverted files and describes a general design tool based on the above technique.

Determining Range Boundaries

One of the more important tasks in creating an inverted file with range attributes is determining the boundaries of the ranges. The primary objective of determining the boundaries is to choose them in such a way that when a user requests records with a specific attribute value or a group of attribute values, a minimum number of excess records is retrieved. There are four basic approaches to achieving this goal.

Suppose the values in the source file are evenly distributed and the probability of using any particular value or range of values is equal. The choice of boundaries would consist of dividing the elements into r ranges. If we had n elements in the source file, each range would consist of ceil(n/r) addresses except for the last range which would contain mod r(n), where ceil is the ceiling function and mod is the modulo function for r.

Suppose the file designer possesses knowledge about the intended retrieval patterns on the attribute values.

In this situation, the choice of boundaries would be based on these patterns. Each range might possess an unique number of addresses and there might be no uniformity to how the ranges are constructed.

Suppose a computer system imposes certain constraints about the amount of information that can be processed at any given moment. In this situation, the file designer might wish to define a minimum, average, and maximum number of addresses that are to be placed in each range.

If the file designer possesses little or no knowlege about the intended retrieval patterns or the distribution of attribute values, a variation of the last method might be appropriate. Under these circumstances, the system could analyze the attribute values and determine the minimum, average, and maximum number of addresses that should be associated with each range. The average value could be determined by taking the square root of the number of distinct attribute values that exist in the file. This forces the number of intervals to be the same as the average number of records to be placed in each interval. The minimum could then be determined by taking half of the average and the maximum could be twice the average.

The last two methods harbor some problems. Suppose a particular attribute value has more records possessing a value than is allowed by the constraints. One possibility would be to eliminate the attribute value. This would be feasible if the cost of maintaining the address list is high for the attribute value in relation to other attributes and the relative usefulness of the attribute in honoring requests. Another possibility would be to allow the range to exist, but to redistribute the previous ranges. The goal of this redistribution is to possibly reduce the number of ranges in the value file. This would occur if the two previous interval could be combined into a single range without violating the maximum number of elements per range and the restriction criteria. If the two intervals could not be combined, then perhaps their addresses can be redistributed to provide two ranges of approximately equal size.

Operations on Range Attribute Inverted Files

The creation of a range attribute inverted file can be accomplished in 3 steps. First, the values and their associated record numbers must be extracted from the source file. Next, the file is sorted so the values are in lexicographic order. This step may be omitted if the source file was already ordered on the values or the desired order was other than lexicographic. For example, the source file values may have been ordered by geographic region rather than lexicographic order.

The next step determines the range boundaries for each of the ranges to be included in the structure. The method employed to accomplish this task is dependent upon the method chosen by the file designer. The basic process involves determining the range boundaries and transferring the appropriate addresses to the address file, making the appropriate entries in the value file, and initializing the differential file.

Satisfying queries with range attribute inverted files is slightly different than searching a standard inverted file. While a standard inverted file typically contains a value entry for each value represented in the source file, the range attribute inverted file contains only endpoints of the ranges. The searching process involves identifying which interval the desired value(s) reside in and retrieving the addresses from the address file and the differential file. Once the addresses are gathered, the corresponding source records are retrieved and examined for the desired characteristics. Note that the retrieval process will retrieve superfluous records since a query may involve a specific attribute value that is a member of some range in the inverted file.

Updating the range attribute inverted file utilizes the same procedure as that for a standard inverted file. The criteria used to determine whether or not the differential file is to be incorporated into the address file is based on the boundary choices made by the designer. The accomplishment of the incorporation involves two scenarios.

Suppose the file designer determined the range boundaries using a priori knowledge about the intended retrieval patterns on the database. The incorporation of the differential file would consist of removing the deleted address entires from the address file and the differential file and then merging the two address lists to create a new address file.

If any of the other possible methods of range boundary determination was used, the inverted directories must be regenerated from the source files. While it is possible to attempt to restructure the inverted files from the existing information, the basic structure of the address files would require change. It would be necessary to include the attribute value with the address in the address file. This would consume an unacceptable amount of space.

CHAPTER IV

A GENERAL DESIGN TOOL FOR RANGE INVERTED FILES

It is possible to construct an inverted index design tool incorporating the previous principles. In addition to providing an environment for creating effective indices, the system should also provide a means for effective update and maintenance of the indices. This chapter describes such a system implemented on a microcomputer.

System Description

The system was created on a Compaq microcomputer utilizing Turbo Pascal, version 2.Ø . The system requires a minimum of 128 K RAM and either 2 floppy disk drives or 1 floppy disk drive and 1 winchester drive. The system consists of three components. Each component can be invoked directly from the operating system of the computer.

The system creates several files for each entry in the directory. Each file resides on the same medium as the source files and is responsible for managing a particular aspect of the directory. The system is designed to work with fixed format files. A fixed format file is one that has a fixed number of characters or bytes for each record in the file. Further, each attribute occupies the same position in each record.

The user must possess a technical description of the value being indexed. This description includes the location of the value in the source record, the data type of the value, and the amount of space consumed by the value. Additionally, it is assumed that the medium has enough space to accomodate the index.

Design Options

There are four basic design options. The user selects the desired design option from a menu. The following describes each design option and the behavior exhibited by the system for that option. See appendix A for sample runs of the tool that describe each option.

User Specified Intervals

This option allows the file designer to specify the boundaries of the ranges. The ranges that are created are unrestricted ranges. The process identifies the current range being defined and prompts the user to provide the low and high values for the range. The low boundary value must be less than or equal to the high boundary value. Once the boundaries have been provided, the system will determine the number of records that possess the values in the range. If any values are skipped from the previous range, the system will inform the user of the number of values skipped. Once the user has been provided with this information, the system allows the user to include the range as defined in the index, exclude the range from the index, or redefine the boundaries of the range.

For example, suppose that it is known that the query patterns on our sample file will consist of three queries: retrieve all salaries less than 2Ø,ØØØ, retrieve all salaries between 2ØØØØ and 39ØØØ, and retrieve all salaries greater than 39ØØØ. In this case, it would be appropriate to define ranges that would retrive these records by extracting a single range from the range inverted index. Figure 7 illustrates the organization for this situation.

	Value File		
Low Key Value	High Key Value	Count	Address
Ø	19999	9	2
2ØØØØ	39ØØØ	3	11
39ØØØ	99999	3	14

Address File

Address: 18 6 14 2 3 4 5 13 15 16 1 7 9 8 1Ø 11 12

Figure 7: User Specified Intervals on Salary

Number of Intervals Specified

This option allows the user to create an inverted file with a fixed number of records per range. If the user specifies r ranges, then each range will have ciel(n/r) elements except for the last group which will have mod k(n) elements, where n is the total number of elements in the source file. The ranges are unrestricted. For example, suppose we decide to create 8 ranges on the salary field. Figure 8 illustrates the resulting value file for the index. Note that the address file will not change.

Value File

Low Value	High Value	Count	Address
15000	15000	2	2
16000	16000	2	4
16000	16000	2	6
16000	16000	2	8
16000	20000	2	10
21000	21000	2	12
22000	40000	2	14
45000	50000	2	16

Figure 8. Number of Intervals Specified on Salary

This option provides fixed size address lists. The designer could choose to make the size of the address lists equal to the size of a physical block of storage.

Auto Specification

This is really two options in one. The user is asked to provide the minimum, average, and maximum number of elements per range. If the user does not provide values for these parameters, the system will calculate values for the user based on the number of distinct attribute values. The user has the option of providing one or all of the parameters for this option. This option creates restricted ranges, i.e., no value can have more than 1 range. In the event the number of records corresponding to a value exceeds the maximum number of elements per range, the designer has the option of including or excluding the range from the index. If the user includes the range in the index, the system will automatically try to combine the two previous intervals. If the intervals cannot be combined, the addresses will be uniformly distributed between the intervals.

	Value File		
Low Value	High Value	Count	Address
15ØØØ	15ØØØ	2	2
16ØØØ	16ØØØ	7	4
2ØØØØ	22ØØØ	4	11
4ØØØØ	5ØØØØ	3	15

Figure 9. Auto Specified Index on Salary

Standard Inverted File

This option allows the user to create a standard inverted file. Each distinct value will have an entry in the value file. The form of the standard inverted file is the same as the form for the range inverted file. It is treated as a special case of the range inverted file where the low and high values for the boundaries happen to be equal for all intervals.

Other Options

Other options are provided that allow the user to view the index files or the intermediate files used in the creation of the directory.

Files Created

Several files are created for each index. The value file contains the values, pointers to the address and differential files, and counts of entries in the address and differential files. The name for this file is provided by the user of the system.

The address file contains the lists of addresses of records corresponding to the values in the value file. The name for this file is the same as the name of the value file with an extension of A.

The differential file contains the addresses of records added to the source file after the index has been created or reorganized. The name for the differential file is DIFF.

The final file created for the directory is the characteristic file. The characteristic file stores information about the indices in the directory. This includes the name of the index, the name of the source file, the location and type of the indexed value, the size of the records in the source file, the type of technique used in the determination of the ranges, and the threshold value for incorporating the differential file in the address file.

Support Modules

There are two support modules for the system. The update module provides a conveneint method for updating information in the directory. The update can be accomplished interactively with the user providing the attribute values and record numbers in addition to the nature of the update operation. Facilities are provided for adding, changing, and deleting entries in the directory.

Another form of update is supported for batch applications. Users can create files that describe the changes that are to be perfomred on the directory. Changes to the directory are limited to adding and deleting entries in the directory.

When adding information to the index, the system will consult the value file and locate the appropriate range. If the range entry has entries in the differential file, the system will insert the new entry at the head of the linked list of entries. If the range does not have any entries in the differential file, the system will update not only the differential file, but also the value file.

When adding entries to the value file, a special situation can arise when a value is being added that is smaller than any value in the value file or the value is greater than any value in the value file. In this situation, the range boundaries must also be modified. For the case when the value being added is smaller than any value, the low value specified for the first range must be updated to reflect the value being added. In the case where the value being added is greater than any value, the high value specified for the high range must be updated to reflect the value being added.

When deleting entries from the value and address files, the system will search for the value in the value file and then traverse the address lists. If it locates the address of the record deleted in the address file, the system will replace the address value with a negative value. If the address of the record deleted was not in the address file, the system will search the differential list for the entry. Again, when the entry is located, the system will replace the address of the record with a negative value.

The final support module determines whether or not the index needs to be restructured. If the threshold value for additions in the directory has been reached, the system will restructure the index. This operation consists of two operations. The deleted entries must be removed from the system and the active entries in the differential file must be incorporated into the address file. The result is a value file and and address file with no deleted entries and a differential file that is empty. The threshold value is typically reached when the differential file has the average number of records for a range entries.

CHAPTER V

PERFORMANCE ANALYSIS

There are two important performance aspects to any computer system. One is the space requirements for the system and the other is the amount of time required to perform various operations. This chapter develops formulae that help categorize the behavior of the system in space and time.

Space Requirements

There are four files used in the directory. They are: value file, address file, differential file, and characteristic file. Each one is responsible for managing one aspect of the directory. Each one is required for the proper functioning of the directory.

The following notation will be used to develop the formulae that describe the space consumption aspect of the index technique. Let:

Lv = number of bytes consumed by the low value of a given range.
Hv = number of bytes consumed by the high value of a given range.
Pt = number of bytes consumed by a pointer to the address of a given record in a given file.
Cv = number of bytes consumed by calculated values such as counts and average, minimum, maximum number of records per range.

Vc = number of bytes consumed by characteristics of of the source value, such as the size of the source records, the size of the value being indexed, and the location of the value in the source record.
Vt = number of bytes consumed by the data type of value being indexed.
Rt = number of bytes consumed by the range type used to create the range inverted index.
Sf = number of bytes consumed by the sort flag.
Fn = The number of bytes consumed by a file name.

Value File

The value file has the following record format: (Low Range Value, High Range Value, Pointer to address file, Address Count, Pointer to differential file, Differential count). This results in the following general formula for the space consumed by one record in bytes:

$Lv + Hv + 2*Pt + 2*Vc$

If we have r ranges in the value file, then

Total Space Consumed = $r*(Lv + Hv + 2*Pt + 2*Vc)$.

For this implementation, the space consumed by Lv and Hv is 16 bytes each. The space consumed by Pt and Vc is 2 bytes each. This yields 4Ø bytes for each value record or r*4Ø bytes for a value file with r ranges.

Address File

The address file consists of records with a single attribute, the address, which is Pt bytes long. If there are n records whose values are indexed, the space consumed by the address file can be expressed as n*Pt bytes. For this implementation, the space consumed by Pt is 2 bytes. This yields n*2 bytes for an address file with n records.

Differential File

This file is responsible for maintaining addresses of records added to the source file since the directory was last reorganized. The file is a linked list with each record possessing the following stucture: (Source record address, Pointer to next in Differential). Each attribute in this file generally consumes Pt bytes, or 2*Pt bytes for each record.

The first record of this file is used to house a pointer to the first available record in the differential file. If there are d additions to the source file, the size of the differential file can be expressed as 2*Pt + d*2*Pt or 2*(Pt + d*Pt) bytes. In this implementation, each Pt consumes 2 bytes yielding 4 + d*4 bytes.

Characterstic File

The characterstic file stores information about the source file and attribute values used to create the directory. Each index in the directory has an entry in the characteristic file. Each record in the characteristic file possesses the following format:

(Source Filename, Index Filename, Source File Record Size, Key Value Size, Key Value Location, Key Data Type, Range Type, Average Number of records per range, Minimum Number of records per range, Maximum Number of records per range, Sort Flag). The source filename and the index filename consume Fn bytes each. The source file record size, attribute value size, and attribute value location each consume Cv bytes. The value type consumes Vt bytes. The range type consumes Rt bytes. The average, minimum, and maximum number of records per range each consume Vc bytes. The sort flag consumes Sf bytes. This yields:

2*Fn + 3*Cv + Vt + Rt + 3*Vc + Sf bytes per record.

If there are i indexes represented in the characteristic file, the total space consumed is:

i*(2*Fn + 3*Cv + Vt + Rt + 3*Vc + Sf)

For this implemenation, Fn consumes 16 bytes, Cv consumes 2 bytes, Vt cosnumes 1 byte, Rt consumes 1 byte, and Sf consumes 1 bytes. This yields total space requirements of i*49 bytes.

Total Space Requirements

The amount of space consumed by each index in the directory can be summarized as follows:

Given:
r - the number of ranges in an index
n - the number of source file records represented in

the index.

d - the number of additions to the source file since the directory was reorganized.

i - the number indices in the directory.

In general, the space, s, consumed by one index is:

$$
\begin{aligned}
s = {} & r*(Lv + Hv + 2*Pt + 2*Vc) + \\
& n*Pt + \\
& 2*(Pt + d*Pt) + \\
& (2*Fn + 3*Cv + Vt + Rt + 3*Vc + Sf)
\end{aligned}
$$

The space consumed by i indices in a directory is:

$$\sum_{j=1}^{i} sj$$

Retrieval Time

Retrieval involves accessing 4 files. The total time required to honor a request is dependent upon the number of records retrieved and is the sum of accessing the four files. All formulae are based on the assumption that the value file is stored in primary storage.

Time to Search Key File

Given r ranges in the value file and assuming the value file's organization is sorted, the time to search the file in primary storage is proportional to log r using a binary search.

Time to Access Address List

Once the value entry is located, the address pointer and count of values is extracted. Since most computer systems access secondary storage devices using a block access scheme, the amount of time required to retrieve the addresses is proportional to the number of blocks that need to be retrieved. This can be expressed as follows:

Given:
b - the number of bytes contained in one block.
t - the number of bytes in the address record.

The blocking factor, B, is

$B = b/t$.

The number of blocks needed to retrieve c records is bounded by:

$\mathrm{ceil}(c/B)$.

Since a block can be retrieved in Ta time, and the address list to be retreived is sequentially ordered, the time required to retrieve the address list is bounded by:

$\mathrm{ceil}(c/B) * Ta$.

Time to Access Differential File

The time required to access a record in the differential file will be longer than accessing information in the address file since each address may reside in a separate block. If we assume d entries in the differential file for a given range attribute, the time required to retrieve the entries has an upper bound of d*Ta.

Time to Access Source File Records

The total number of records to be retrieved from the source file is equal to c + d . The amount of time required to access the records has an upper bound of (c+d)*Ta since each record may reside in a different block.

Total Time to Satsify Query

The total time required to satisfy a simple query is the sum of the accesses to the 4 files. This can be expressed as follows:

Given:
B - blocking factor
c - number of addresses from address file
d - number of addresses retrieved from diff. file
r - number of ranges in index
Ta - time required to access a block from secondary storage

Total time to satisfy a simple query
log r + (int (c/B) + c + 2d) * Ta

Example

Given:
A file with 2000000 records.
An index on a value that has 250000 distinct values.
The values are evenly distributed among the records of the file.
There is no particular order to the values in file.
A range inverted index is created using the auto-specification option.
No need for differential file since values are static and no new entries are committed.
Each block houses 512 bytes of data.

The following range parameters result:

Average Number of records = SQRT(250000)=500
Minimum Number of records = 1/2 * 500 = 250
Maximum Number of records = 2 * 500 = 1000
Records per value = 2000000/250000 = 8
Values per range = ciel(500/8) = 63
Addresses per range = 63 * 8 = 504
Number of ranges = 250000/63 = 3968

For a standard inverted file the folowing parameters result:

250000 value entries
8 addresses per value

Space Consumption of Indexes (This implementation)

Range Inverted
Value File = 3968 * 40 = 143K
Address File = 250000 * 2 = 500K
TOTAL 643K

Standard Inverted
Value File = 250000*(16 + 2 + 2) = 5000K
Address File = 250000 * 2 = 500K
TOTAL 5500K

Time Required to Locate Records with Single Value

Range Inverted

Search of Value File - Bounded by log 3968 = 12
@ Ø.ØØØ1 sec/access = .ØØ12 sec

Retrieval of Addresses - 1 block required
@ Ø.ØØ1 sec/acces = .ØØ1 sec

Retrieval and Examine Source - Bounded by 5Ø4 * access
@ Ø.ØØ1 sec/access = .5Ø4 sec

TOTAL is bounded by .5Ø62 sec.

Standard Inverted

Search of Key value File - Bounded by log 25Ø,ØØØ = 21
@ Ø.ØØ1 sec/access = .Ø21 sec

Retrieval of Addresses - 1 block required
@ Ø.ØØ1 sec/access = .ØØ1 sec

Retrieval of Source - Bounded by 8 * access
@ Ø.ØØ1 sec/access = .ØØ8 sec

TOTAL is bounded by .Ø3Ø sec.

While the standard inverted file will outperform the range inverted file in this situation, the standard inverted consumes almost 1Ø times as much space. Note further that the range inverted index can be housed in primary storage while the standard inverted must remain on secondary storage due to its enormous size. For most computer systems, the amount of space versus the increase in retrieval time is an uequal trade of space for time. The next example, however, demonstrates how the range inverted file can outperform the standard inverted file in both time and space.

Time Required to Locate a Range of Records

Assume that we wish to retrieve a single range of values that exists in a single range in the range inverted file.

Range Inverted

Search of Value File - Bounded by log 3968 = 12
@ Ø.ØØØ1 sec/access = .ØØ12 sec

Retrieval of Addresses - 2 accesses
@ Ø.ØØ1 sec/access = .ØØ2 sec

Retrieve Source - Bounded by 5Ø4 accesses
@ Ø.ØØ1 sec/access = .5Ø4 sec

TOTAL is bounded by .519 sec

Standard Inverted

Search of Value File - Bounded by 8*log 25Ø,ØØØ = 168
@ Ø.ØØ1 sec/access = .168 sec

Retrieval of Addresses - 8 accesses
@ Ø.ØØ1 sec/access = .ØØ8 sec

Retrieval of Source - Bounded by 5Ø4 accesses
@ Ø.ØØ1 sec/acces = .5Ø4 sec

TOTAL is bounded by .68Ø sec

Notice that the difference in time from the first situation to the second situation can be explained by the retrieval of superfluous records in the range inverted situation. When satisfying queries that specify several values, the range inverted file will outperform the standard inverted file in most situations. Also, it should be noted that the space requirements for the range inverted file is an order of magnitude less than the storage requirements for the standard inverted file.

Index Creation

There are several distinct steps in the creation of the index. The first step for all indexes is to extract the values and corresponding record addresses. This step can be accomplished with one pass through the source file. The time required to accomplish this task is O(n) given n records in the source file.

The second step usually involves sorting the values. The amount of time rquired to accomplish this task is O(n log n).

The third step involves the creation of a standard inverted file. This requires one pass through the sorted file and the amount of time required to accomplish this task is O(n).

Thus, the amount of time required to prepare the values for processing is O(n log n) if sorting is involved or O(n) if no sorting is involved.

Finally, the amount of time to create the index is O(n) since all methods require a single pass through the work files. THe total time required to create a index is therefore O(n log n).

Conclusion

This paper has presented a technique that is particularly useful in creating general direcotries. The algorithms described provide file designers with a comprehensive set of tools for the creation and maintenance of indices in the directories. Unlike most systems which give a file designer only a single type of indexing scheme, this tool provides the designer with 4 schemes. The methods have been applied to a wide variety of data types and retrieval patterns. The system and method provide an effective general directory.

APPENDIX A

Algorithms

ALGORITHMS

```
MAIN PROCEDURE

  LOOP
    DISPLAY OPTIONS
    GET KEY VALUE DESCRIPTION
    EXTRACT KEY VALUES AND RECORD ADDRESSES
    IF USER WANT TO SORT THE DIRECTORY THEN
      SORT WORK FILE
    ENDIF
    CREATE STANDARD INVERTED FILE
    CREATE SPECIFIED INDEX TYPE
  UNTIL EXIT

END MAIN

CREATE USER SPECIFIED DIRECTORY PROCUDURE

  READ FIRST ENTRY FROM STANDARD INVERTED FILE
  INITIALIZE INTERVAL COUNT, VALUES SKIPPED
  WHILE NOT EOF AND MORE INTERVALS DO
    ACCEPT LOW AND HIGH BOUNDARIES FOR NEXT INTERVAL
    WHILE KEY VALUE <= HIGH BOUNDARY DO
      IF LOW BOUNDARY <= KEY VALUE THEN
        INCREMENT COUNT BY 1
      ELSE
        INCREMENT VALUES SKIPPED BY 1
      ENDIF
      READ NEXT ENTRY FROM STANDARD FILE
    END WHILE
    DISPLAY INTERVAL COUNT AND VALUES SKIPPED TO USER
    CASE DECISION OF
      I: INCLUDE INTERVAL
      R: REDEFINE INTERVAL
      E: EXCLUDE INTERVAL
    END CASE
    INTIALIZE NEW INTERVAL COUNT, VALUES SKIPPED
  END WHILE

END CREATE USER SPECIFIED PROCEDURE
```

```
CREATE NUMBER OF INTERVALS PROCEDURE

  GET NUMBER OF INTERVALS FROM USER
  COMPUTE NUMBER OF RECORDS/INTERVAL = TOTAL RECORDS/
    NUMBER OF INTERVALS
  FOR I=1 TO NUMBER OF INTERVALS DO
    COPY NUMBER OF RECORDS/INTERVAL RECORDS FROM
      STANDARD INVERTED FILE
    STORE CORREPSONDING KEY VALUE RECORD IN INDEX
  END FOR
END CREATE NUMBER OF INTERVALS PROCEDURE

CREATE OPTIMIZED PROCEDURE

(INTERVAL COUNT refers to the number of records in the
  current interval.
 VALUE COUNT is the number of records for the current
  value)

  GET AVERAGE NUMBER OF RECORDS/INTERVAL FROM USER
  IF NONE SUPPLIED THEN
    COMPUTE AVERAGE NUMBER = SQUARE ROOT (TOTAL DISTINCT
      ATTRIBUTES)
  ENDIF
  GET MINIMUM NUMBER OF RECORDS/INTERVAL FROM USER
  IF NONE SUPPLIED THEN
    COMPUTE MINIMUM = AVERAGE NUMBER / 2
  ENDIF
  GET MAXIMUM NUMBER OF RECORDS/INTERVAL FROM USER
  IF NONE SUPPLIED THEN
    COMPUTE MAXIMUM = AVERAGE NUMBER * 2
  ENDIF
  READ FIRST ENTRY FROM STANDARD INVERTED
  INITIALIZE FIRST INTERVAL
  WHILE NOT EOF DO
    WHILE ((VALUE COUNT + INTERVAL COUNT) < AVERAGE) AND
      (NOT EOF) DO
      ADD VALUE COUNT TO INTERVAL COUNT
      READ NEXT ENTRY FROM STANDARD INVERTED
    END WHILE
    IF NOT EOF THEN
      IF VALUE COUNT + INTERVAL COUNT <= MAXIMUM THEN
        ADD VALUE COUNT TO INTERVAL COUNT
        STORE CURRENT INTERVAL
      ELSE
        STORE INTERVAL EXCLUDING CURRENT ENTRY
        IF VALUE COUNT > MAXIMUM THEN
          INFORM USER OF EXCEPTION
          GET DECISION
          IF STORE THEN
            CALL COMBINE OR REDISTRIBUTE
          ENDIF
        ENDIF
      ENDIF
      READ NEXT EXTRY FROM STANDARD INVERTED
      INITIALIZE NEW INTERVAL
    ELSE
      STORE LAST INTERVAL
    ENDIF
  END WHILE
END CREATE OPTIMIZED PROCEDURE
```

```
COMBINE OR REDISTRIBUTE PROCEDURE

  ( Current interval is I+1, while I and I-1 are two
    previous intervals.)

IF INTERVALS I AND I-1 ARE CONSECUTIVE THEN
    SUM = INTERVAL COUNT(I) + iNTERVAL COUNT(I-1)
    IF SUM > MAXIMUM THEN
      READ FIRST ENTRY OF INTERVAL I-1
      INITIALIZE INTERVAL I-1
      NEWDIFF = SUM - INTERVAL COUNT (I-1)
      REPEAT
        READ NEXT STANDARD KEY VALUE ENTRY
        ADD VALUE COUNT TO INTERVAL COUNT (I-1)
        OLDIFF=NEWDIFF
        NEWDIFF=ABS(INTERVAL COUNT(I) - INTERVAL
          COUNT(I-1))
      UNTIL NEWDIFF > OLDIFF
      TRANSFER LAST ENTRY OF TOTAL VALUES FILE TO
        INTERVAL I
      STORE ENTRIES FOR INTERVALS I, I-1
    ELSE
      COMBINE INTERVALS I, I-1
    ENDIF
  ENDIF
END COMBINE OR REDISTRIBUTE

UPDATE PROCEDURE

( interactive updates include add,change, or delete
  while batch updates only allow the add and delete
  options. )

  GET INDEX TO UPDATE FROM USER
  GET METHOD OF UPDATE FROM USER
  IF INTERACTIVE UPDATE THEN
    DONE=FALSE
    WHILE NOT DONE DO
      GET UPDATE OPERATION FROM USER
      CASE OPERATION OF:
        ADD: DO ADD PROCEDURE
        CHANGE: DO CHANGE PROCEDURE
        DELETE: DO DELETE PROCEDURE
        EXIT: DONE=TRUE
      ENDCASE
    ENDWHILE
  ELSE
    GET BATCH FILE NAME FROM USER
    WHILE MORE BATCH UPDATE RECORDS DO
      CASE OPERATION OF
        ADD: DO ADD PROCEDURE
        DELETE: DO DELETE PROCEDURE
      ENDCASE
    ENDWHILE
  ENDIF
END UPDATE PROCEDURE
```

```
ADD PROCEDURE

  LOCATE KEY VALUE IN KEY ADDRESS FILE
  IF KEY VALUE NOT IN EXISTING RANGE THEN
    UPDATE KEY VALUE RECORD
  ENDIF
  GET FREE POINTER FROM DIFFERENTIAL FILE
  UPDATE DIFFERENTIAL FILE FREE POINTER
  IF DIFFERENTIAL POINTER > Ø THEN
    DIFFERENTIAL  POINTER  OF  NEW  RECORD = DIFFERENTIAL
      POINTER OF VALUE RECORD
    DIFFERENTIAL  POINTER  OF  VALUE  RECORD  =  NEW
      DIFFERENTIAL RECORD
    CHANGE POINTER OF CURRENT RECORD TO NEW RECORD
    ADD RECORD TO DIFFERENTIAL FILE
  ELSE
    UPDATE DIFFERENTIAL POINTER IN KEY VALUE FILE
    ADD RECORD TO DIFFERENTIAL FILE
  ENDIF
END ADD PROCEDURE

CHANGE PROCEDURE

  GET OLD VALUE AND RECORD ADDRESS FROM USER
  CALL DELETE PROCEDURE
  GET NEW VALUE FROM USER
  CALL ADD PROCUDURE
END CHANGE PROCEDURE

DELETE PROCEDURE

  LOCATE KEY VALUE
  POSITION ON FIRST ADDRESS RECORD IN ADDRESS FILE
  DONE = FALSE
  FOUND = FALSE
  COUNT=1
  WHILE NOT DONE DO
    IF ADDRESS RECORD = ONE TO DELETE THEN
      CHANGE ADDRESS TO NEGATIVE VALUE
      DONE=TRUE
      FOUND=TRUE
    ELSE
      GET NEXT RECORD
      ADD 1 TO COUNT
      IF COUNT > NUMBER OF ADDRESSES THEN
        DONE=TRUE
      ENDIF
    ENDIF
  ENDWHILE
  IF NOT FOUND THEN
    PROCESS DIFFERENTIAL FILE
  ENDIF
END PROCEDURE DELETE
```

```
RESTRUCTURE PROCEDURE

  ASK USER FOR INDEX TO CHECK
  OPEN CHARACTERISTIC FILE
  RETRIEVE INFORMATION FOR INDEX
  OPEN KEY VALUE FILE FOR INDEX
  SUM = ZERO
  WHILE MORE ENTRIES IN KEY VALUE FILE DO
    SUM = SUM + COUNT IN DIFFERENTIAL
  ENDWHILE
  IF SUM > THRESHOLD THEN
    IF USER SPECIFIED RANGES THEN
      MERGE ADDRESS LISTS
    ELSE
      RESTRUCTURE INDEX
    ENDIF
  ELSE
    INFORM USER INDEX DOES NOT NEED TO BE RESTRUCTURED
  ENDIF
END RESTRUCTURE PROCEDURE


EXTRACTION PROCEDURE

  GET FIRST DATA BLOCK
  LOW BYTE = 1
  HIGH BYTE = BUFFER SIZE
  FOR I =1 TO NUMBER OF RECORDS DO
    START BYTE = RECORD # * RECORD LENGTH + KEY LOCATION
    IF START BYTE > HIGH BYTE THEN
      GET NEXT DATA BLOCK
      ADJUST LOW AND HIGH BYTES
    ENDIF
    END BYTE = START BYTE + KEY LENGTH
    IF END BYTE > HIGH BYTE THEN
      DIFF = HIGH BYTE - START BYTE
      TRANSFER DIFF BYTES TO WORK AREA
      GET NEXT DATA BLOCK
      ADJUST LOW AND HIGH BYTES
      DIFF = KEY LENGTH - DIFF
      TRANSFER DIFF BYTES TO WORK AREA
    ELSE
      TRANSFER KEY LENGTH BYTES TO WORK AREA
    ENDIF
    CONVERT DATA TO STANARD REPRESENTATION
  ENDFOR
END EXTRACTION PROCEDURE
```

APPENDIX B

SAMPLE RUN

SAMPLE RUN

```
                    DIRECTORY UTILITY

                           BY

                    Edward J. Peeler

Do You Need Help? n

Do you wish to log the session (y/N)? y

DESIGN OPTIONS

1 - You Specify the Ranges
2 - You Specify the Number of Ranges
3 - System Creates Optimal Intervals
4 - Standard Inverted File
5 - Help!
6 - Display Standard Inverted File
7 - Display Resultant File
8 - Exit System

Option #? 1

              SOURCE FILE CHARACTERISTICS

File Name - indata.dat
Record Length in Bytes - 36
Number of Records in File - 100
Starting Byte of Attribute - 22
Data Type of Attribute (I=integer,R=Real,S=String) - i

Index Name - ind1
```

```
                    USER SPECIFIED INTERVALS

Interval 1
Low Value - 1
High Value - 2Ø6
Number of Records in Interval: 18 Ø Value(s) skipped
I)nclude/E)xclude/R)edfine Interval? (Default=I) i

                    USER SPECIFIED INTERVALS

Interval 2
Low Value - 211
High Value - 3Ø52
Number of Records in Interval: 54 Ø Value(s) skipped
I)nclude/E)xclude/R)edefine Interval? (Default=I) i

                    USER SPECIFIED INTERVALS

Interval 3
Low Value - 3Ø56
High Value - 5Ø75
Number of Records in Interval: 28 Ø Value(s) skipped
I)nclude/E)xclude/R)edefine Interval? (Default=I) i

                         DESIGN OPTIONS

1 - You Specify Ranges
2 - You Specify the Number of Ranges
3 - System Creates Optimal Intervals
4 - Standard Inverted File
5 - Help!
6 - Display Standard Inverted File
7 - Display Resultant File
8 - Exit System

Option #? 7

Low Value      High Value      Count   Pointer   DCOUNT   DPTR
_______________________________________________________________
     1            2Ø6            18       Ø         Ø       Ø
   211           3Ø52            54      18         Ø       Ø
  3Ø56           5Ø75            28      72         Ø       Ø
---------------------------------------------------------------
DISPLAY COMPLETE - PRESS RETURN TO CONTINUE
                         DESIGN OPTIONS

1 - You Specify Ranges
2 - You Specify the Number of Ranges
3 - System Creates Optimal Intervals
4 - Standard Inverted File
5 - Help!
6 - Display Standard Inverted File
7 - Display Resultant File
8 - Exit System

Option #? 2
```

```
                SOURCE FILE CHARACTERISTICS

File Name - indata.dat
Record Length in Bytes - 36
Number of Records in File - 100
Starting Byte of Attibute - 22
Data Type of Attribute (I=Integer,R=Real,S=String) i

Index Name - ind2

           USER SPECIFIED NUMBER OF INTERVALS

Number of Intervals 12

                     DESIGN OPTIONS

1 - You Specify Ranges
2 - You Specify the Number of Ranges
3 - System Creates Optimal Intervals
4 - Standard Inverted File
5 - Help!
6 - Display Standard Inverted File
7 - Display Resultant File
8 - Exit System

Option #? 7

Low Value     High Value     Count  Pointer  DCOUNT  DPTR
---------------------------------------------------------
     1           103           9       0        0      0
   106           206           9       9        0      0
   211           250           9      18        0      0
   250           314           9      27        0      0
   815          1270           9      36        0      0
  2013          2112           9      45        0      0
  2212          2422           9      54        0      0
  3017          3052           9      63        0      0
  3056          3510           9      72        0      0
  3510          4039           9      81        0      0
  4043          5075           9      90        0      0
  5075          5075           1      99        0      0
---------------------------------------------------------
DISPLAY COMPLETE - PRESS RETURN TO CONTINUE
                     DESIGN OPTIONS

1 - You Specify Ranges
2 - You Specify the Number of Ranges
3 - System Creates Optimal Intervals
4 - Standard Inverted File
5 - Help!
6 - Display Standard Inverted File
7 - Display Resultant File
8 - Exit System
```

```
Option #? 3

                    SOURCE FILE CHARACTERISTICS

File Name - indata.dat
Record Length in Bytes - 36
Number of Records in File - 1ØØ
Starting Byte of Attibute - 22
Data Type of Attribute (I=Integer,R=Real,S=String) - i

Index Name - ind3

                       UNIFORM OPTIMIZATION

Average Number of Records per Interval
Average Number of Records - 9
Minimum Number of Records per Interval
Minimum Number of Records per Interval - 4
Maximum Number of Records per Interval
Maximum Number of Records per Interval - 18

                          DESIGN OPTIONS

1 - You Specify Ranges
2 - You Specify the Number of Ranges
3 - System Creates Optimal Intervals
4 - Standard Inverted File
5 - Help!
6 - Display Standard Inverted File
7 - Display Resultant File
8 - Exit System

Option #? 7

Low Value     High Value     Count    Pointer   DCOUNT DPTR
-----------------------------------------------------------
     1           1Ø6           1Ø      Ø          Ø     Ø
   1Ø7           211           1Ø      1Ø         Ø     Ø
   212           26Ø           1Ø      2Ø         Ø     Ø
   261          1Ø2Ø           1Ø      3Ø         Ø     Ø
  1Ø24          2Ø5Ø           1Ø      4Ø         Ø     Ø
  2Ø7Ø          24Ø7           1Ø      5Ø         Ø     Ø
  2419          3Ø5Ø           11      6Ø         Ø     Ø
  3Ø52          351Ø           13      71         Ø     Ø
  3616          4Ø64           1Ø      84         Ø     Ø
  4Ø68          5Ø75            6      94         Ø     Ø
-----------------------------------------------------------
DISPLAY COMPLETE - PRESS RETURN TO CONTINUE

                          DESIGN OPTIONS

1 - You Specify Ranges
2 - You Specify the Number of Ranges
3 - System Creates Optimal Intervals
```

```
4 - Standard Inverted File
5 - Help!
6 - Display Standard Inverted File
7 - Display Resultant File
8 - Exit System

Option #? 8
```

BIBLIOGRAPHY

Cardenas, A.F.: "Analysis and Performance of Inverted Database Structures.", Comm. ACM, 18, No. 5 (1975), 253-63.

Conrad, L.; Bloom, S.; Cooper, C.; Cannon, T.; Friedman, R.H.; Horowitz, J.; Krikorian, J.; Lopez, J: "The Cancer Data Management System Statistics Package.", Proceedings of the Fourth Annual Symposium on Computer Applications in Medical Care, 198Ø, 1281-5.

Fedorowics, J.: "A Zipfian Model of an Automatic Bibliographic System: An Application to MEDLINE." J. Am Soc Inf Sci, 33, No. 4 (1982), 223-32.

Gersting, J.M. Jr.;Conneally, P.M.; Rogers, K.; Blum, B.I.: "Two Search Techniques within a Human Pedigree Database.", Proceedings of the Sixth Annual Symposium on Computer Applications in Medical Care, 1982, 842-6.

Hardings, A.F.; Willet, P.W.: "Matrices" J. Am Soc Inf Sci, 31, No. 4 (198Ø), 298-3ØØ.

Hoffer, J.A.: "Database Design Practices for Inverted File." Information and Management, 3, No. 4 (198Ø) 149-61.

Horowitz, E.; Sahni, S: Fundamentals of Data Structures, New York: Computer Science Press, 1976.

Jakobsson, M.: "Reducing Block Accesses in Inverted Files by Partial Clustering." Inf Systems, 5, No. 1 (198Ø), 1 - 5.

Jakobsson, M.; Nevalainen,D.: "On the Organization of Hybrid Indexes.", Proceedings of the International Conference on Databases, 198Ø, 25Ø-9.

Jakobsson, M: "Evaluation of a Hierarchical Bit Vector Compression Techique." Information Processing Letters, 14, No. 13 (1982), 147-9.

Johnson, J. S. and Webster, E: "Updating an Inverted File Index - A Performance Comparison of Two Techniques", Computer Journal, 25 (1982), 169-75.

Knuth, D.E.: The Art of Computer Programming Vol. 3; New York: Addison-Wesley, 1973.

Lie, J.W.: "Algorithms for Parsing Search Queries in Systems with Inverted Files" ACM Tran Database Systems, 1, No. 4 (1976), 299-316.

Motzkin, D.: "The Use of Normal Multiplication Tables for Information Storage and Retrieval" Comm ACM, 2, No. 3 (1979), 193-207.

Motzkin, D; Williams, K; Chang K: "Uniform Organization of Inverted Files.", Proceeding NCC-84, 1984, 567-85.

Nevalainen, D.; Jakobsson, M.; Berg, G.: "Compression of Clustered Inverted Files" Mathematial Foundations of Computer Science (1978), 219-225.

Putkanen, A.: "The Order of Merging Operations for Queries in Inverted File Systems" Int J. Cmputer and Inf Sci, 9, No. 5 (1980).

Schkolnick, M: "A Survey of Physical Database Design Methodology and Technique.", Proceedings Fourth International Conference on Very Large Databases, 1978,474-87.

Schultheisz, R.J.: "Toxline Evolution of an On-Line Interactive Bibliographic Database" J Am Doc Inf Sci, 32, No. 6 (1981).

Togasi, M. and Tanaka, K.: "An Information Management System for Charged Particle Nuclear Reaction Data" J Inf Sci Prin and Prac, 4, No. 5 (1982), 213-23.

Tuttle, M.S.; Sheretz, A.; Bloise, M.; Nelson, D.: "Expertness from Structured Text, RECONSIDER: A Diagnostic Prompting Program.", Proceedings of the Conference on Applied Natural Langauge Processing, Association of Computational Linguistics, 1983, 124-31.