# STORAGE STRUCTURES FOR RELATIONAL DATA BASE MANAGEMENT SYSTEMS

by

Gerald David Held

Memorandum No. ERL-M533

11 August 1975

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Storage Structures for Relational Data Base Management Systems

Ph.D. Gerald David Held EECS

Signature_____
Chairman of Committee

ABSTRACT

Storage structures are examined which allow efficient access to information in a relational data base management system. The major areas investigated are: 1) storage structures for data relations, 2) storage structures for auxiliary information to speed access to data, and 3) a strategy for selecting structures based on query statistics.

First, a large class of possible storage structures for data relations is examined. A generalized directory structure is defined and is shown to provide better performance than either normal directories or simple order preserving functions. An algorithm for constructing generalized directories is described with complexity which is linear in file size and results of experiments using the algorithm are given. Tradeoffs between dynamic directories (i.e. continuously reorganizing) and static directories (i.e. periodically reorganizing) are discussed. Static directories are shown to be preferable on the basis of secondary index, concurrency, and directory size considerations. Next, several types of auxiliary storage structures are considered. Secondary indices on functions of attributes are introduced and a method for reusing aggregation information is

presented. Finally, a general strategy for making storage structure choices is presented. The query model previously used for key selection is extended to provide more accurate choice of key domains. The strategy selects data relation storage structures, primary key domains and auxiliary structures.

## ACKNOWLEDGMENTS

It is my pleasure to acknowledge the many contributions and continual encouragement that I have received from Professor Stonebraker. I am also grateful to Professor Wong for his suggestions and counseling and to Professor Maron for reading the manuscript. In addition, thanks are due to my fellow students on the INGRES project who are making relational data base systems a reality.

This work would not have been undertaken if it were not for the support I have received from my friends at RCA in obtaining the David Sarnoff Fellowship.

Finally I would like to thank Constance, Jennifer, and Jessica, for their encouragement and understanding.

Partial research sponsored by the U.S. Army Research Office Durham Grant DAHC04-74-G0087 and the National Science Foundation Grant DCR75-03839.

# TABLE OF CONTENTS

# CHAPTER 1

# Introduction

## 1.1 Evolution of Data Base Management

Data base management systems are currently undergoing an evolution similar to that which programming languages and operating systems have previously gone through. This evolution consists of understanding the basic issues in the field, analyzing tradeoffs of different solutions and designing general algorithms to solve the problems. Each evolution has begun with individual, ad hoc solutions to problems in a way specific to the problem at hand. As similarities between many problems are discovered, more basic and general algorithms are devised. As the performance of the general algorithms is improved and approaches that of the specific solutions, the general techniques are more frequently applied. This evolution toward more general tools for problem solving allows for easier and less complicated solutions at an occasional cost of computer time. The evolution of data base management systems has progressed to the point where systems exist (such as [CODA71a]) to aid in data definition and low level procedure writing. However the processing of user queries into operations on the data base is still an ad hoc procedure specific to the problem at hand. Recently, several systems have been proposed

for specifying general queries at a high level. These systems propose to provide general procedures for processing the queries. It is the purpose of this dissertation to examine some of the issues involved in making such general data base management systems approach the efficiency of problem specific solutions.

## 1.2 Data Model

Of the many possible data models, the three most widely considered for use in data base management systems are the network model [CODA71a], the hierarchic model [IBM70], and the relational model [CODD70]. There has been considerable debate as to which of these models should be used [CODD74, DATE74, WHIT74, BACH74, SIBL74, LUCK74, STON75]. We have chosen the relational model for three reasons. First, a major argument against the relational model has been that it could not be implemented efficiently. It is our desire to investigate this charge by examining one of the major implementation problems of such a system. Second, this model offers a simple and uniform view of data to the user. And third, the relational model makes a clear separation between the user's data model and the underlying storage structures thus making automatic storage structure selection feasible. It is important to note that this work can, for the most part, be applied to a data base system which uses a network or hierarchic data model as long as there is a clear separation between the user's data model and the actual storage structures which implement that

model.

Formally we define the relational model as follows. Given sets $D_1,\ldots,D_N$ (not necessarily distinct) a RELATION $R(D_1,\ldots,D_N)$ is a subset of the Cartesian product $D_1x\ldots xD_N$. In other words, R is a collection of N-tuples $X = (X_1,\ldots,X_N)$ where $X_i$ is an element of $D_i$ for i in {1,...,N}. The sets $D_1,\ldots,D_N$ are called DOMAINS of R and R has DEGREE N. The only restriction put on relations is that they be normalized. Hence, every domain must be SIMPLE, i.e. it cannot have members which are themselves relations.

Clearly, R can be thought of as a table with elements of R appearing as rows and with columns labeled by domain names as il-

employee relation

| tuple | name | dept | salary | manager | birth | start |
|---|---|---|---|---|---|---|
| 1 | Adams | candy | 12000 | Baker | 1939 | 1965 |
| 2 | Baker | admin | 20000 | Harding | 1927 | 1955 |
| 3 | Harding | admin | 31000 | none | 1917 | 1949 |
| 4 | Johnson | toy | 14000 | Harding | 1946 | 1966 |
| 5 | Jones | toy | 14000 | Johnson | 1943 | 1968 |
| 6 | Smith | toy | 10000 | Jones | 1950 | 1970 |

Figure 1.1 A Sample Relation

lustrated in figure 1.1. The figure indicates an EMPLOYEE relation with domains NAME, DEPT, SALARY, MANAGER, BIRTH year, and START year. Each employee has a manager (except for Harding who is presumably the company president), a salary, a birth year, a start year, and is in a department.

Each column in a tabular representation for R can be thought of as a function mapping R into $D_i$. These functions will be called ATTRIBUTES. An attribute will not be separately designated but will be identified by the domain defining it. For a more detailed discussion of the relational model see [CODD70, CODD71b, CODD72a].

## 1.3 Query Language

Although the work described here is, for the most part, applicable to any relational data base system, it is convenient to consider questions which arise in the discussion in terms of a specific system and query language. The system chosen is INGRES [HELD75a] and its query language, QUEL, which have been developed at the University of California, Berkeley. At this point, the reader who is unfamiliar with QUEL is advised to refer to Appendix A where a short description of QUEL is given along with a few examples.

## 1.4 Implementation Issues

The construction of a high level data base management system involves several building blocks. Figure 1.2 indicates the pieces which are used in INGRES and are typical for such a system. These pieces are now briefly described in the order that a query is processed. First is the query formulation box which

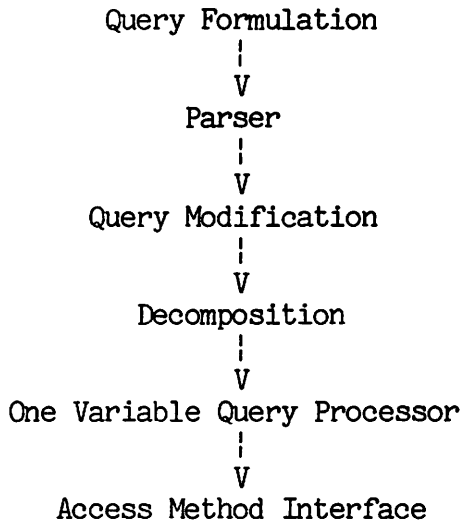


Figure 1.2 INGRES Implementation

provides a method of posing queries. This function may take the form of an interactive text editor, a graphics interface [MCDO75], an interactive English-like language [CODD74a] , or it may be a part of a host programming language. The next part of the system is a parser which recognizes correct queries and converts them to a more convenient form for further processing. The following processing step is query modification. Here high level protection and integrity constraints are added [STON74,STON74d] and queries on virtual relations [CHAM75] are changed to queries on real relations. At this point the actual data retrieval or update request is understood and processing of the request begins. Several approaches have been suggested for breaking down the, possibly very complex, query and arriving at the required answer [MCDO74, ASTR74, SMIT75, PECH75]. The approach taken in INGRES is called decomposition [WONG75] and is described briefly in Appendix B. In this approach, queries are broken down into

successively simpler queries until a point is reached where a query involves only a single relation. At this point, an access processor actually retrieves tuples from the relation, tests to see if they meet the qualification, and displays the results or makes the requested updates. To access and update the relation, the access monitor makes use of a set of access methods (see Appendix C) which provide a relational view of data and support a variety of actual storage structures. It appears that the two major areas in the above description which have the most effect on the efficiency of the data base system are decomposition and choice of storage structures. The research described here deals with the second of these major problems.

## 1.5 Overview of the Dissertation

The main goal of this research is to explore the ways in which storage structures for data can be chosen wisely in order to provide a relational view of data to the user while providing fast access and update capabilities to the relational data base management system. To this end, we have divided the problem into the following the parts: 1) Choosing a set of storage structures for data relations, 2) Choosing a set of auxiliary information structures, and 3) Defining a strategy for making storage structure decisions. The dissertation is organized as follows. Chapters 2 and 3 are concerned with choosing a good set of storage structures to implement data relations. Chapter 2 in-

troduces the terminology and storage model used in the rest of the dissertation and examines several issues in determining the usefulness of storage structures. Chapter 3 considers several classes of storage structures and indicates the usefulness of each. Generalized directories are introduces and are shown to provide better performance than either normal directories or simple order preserving functions. An algorithm for constructing generalized directories is described and results of experimental use of the algorithm are presented. Static directories are shown to be preferred to dynamic directories on the basis of secondary index, concurrency, and directory size considerations. Chapter 4 considers what redundant information might usefully be stored to speed access to data. This redundant data includes secondary indices (inversions) and other information which is used in processing a query. Secondary indices on functions of attributes are introduced and a method is presented for reusing aggregation information. Chapter 5 presents a general strategy for choosing storage structures including which domains should be primary keys and which domains (if any) should be secondary keys. The strategy presented builds on previous work in the area of secondary index selection. The query model previously used is modified to provide better choice of secondary keys and is then extended to handle primary key selection, and primary structure choice. Also in this chapter, a method is described for obtaining the parameters needed for the selection process. Chapter 6 discusses conclusions and directions for future research.

# CHAPTER 2

## Storage Structure Considerations

We will now describe a model for secondary storage and define the terminology used throughout the dissertation. The remainder of this chapter will examine several considerations which are important in choosing a storage structure. These considerations will form the basis of our investigation of a useful set of storage structures in the next chapter. The discussion in this and the next chapter will be concerned with structures for primary data storage whereas issues involved with auxiliary data storage (i.e. secondary indices) will be considered in chapter 4.

### 2.1 Storage Structure Model

In data base management systems, the stored data is of such large volume and of such long lifetime that it is only economical to store the data on low cost storage devices. We refer to these devices as "secondary storage" as opposed to the faster "main storage" which is used primarily as a temporary storage area during actual data processing. Depending on the actual hardware involved in a system, certain types of devices may be classified as main or secondary. We will not attempt to classify devices into one category or the other except to say that at the time of this writing, main storage typically consists of devices like

fast registers, cache memories, monolithic memories, and core memories. Secondary storage normally includes disks, drums, and magnetic tapes. As technology changes, different devices will be included in each category, however, as technology improves, there will probably always be a significant difference in cost, size, and performance between some set of main storage devices and a different set of secondary storage devices. By the nature of the data in a data base system, it will almost always reside on secondary storage with portions of it being transferred to main storage for processing. We refer to the basic quantity of data transferred between main and secondary storage as a "page". A major assumption throughout this work is that the transfer of pages between main and secondary memory is costly and that by the nature of data base processing, this page transfer time will be the dominant cost with actual computation time being small in comparison. It is therefore the goal in data base storage structure selection to choose methods of storing data on secondary memory that will tend to minimize the number of page transfers which occur. This remains a valid goal as long as there continues to be orders of magnitude difference in speed between main and secondary memory and as long as data base processing continues to involve a high ratio of data search to cmputation.

There are many ways to classify all of the possible data structures which might be used to store relations on secondary memory. We first divide all storage structures into two classes, keyed structures and non-keyed structures. A keyed structure is one in

which a domain (or combination of domains) of a tuple is used to determine where in secondary storage the tuple should be stored. This domain is called the "primary storage structure key", "primary key", or simply, the "key". For instance in the EMPLOYEE relation, NAME might be the key domain. In such structures, when a value of the key domain is specified (i.e. NAME = "Jones"), the tuple(s) having the specified value can be located directly without a full scan of the relation. On the other hand, a non-keyed structure is one in which the tuples are stored using some criteria which is independent of the value of the tuple. Examples of non-keyed structures are stacks, queues, and unordered tables.

Non-keyed structures do not provide any ability to limit the number of tuples examined when specific values of one or more domains are supplied. As a result of this characteristic, non-keyed structures are only of interest for special purpose functions in a relational data base system. Such special purposes may include moving data back and forth between the data management system and other programs, storing temporary or intermediate relations, and other situations which will require full passes over the relation. We will therefore confine our investigation to storage structures which store tuples as a function of one or more key domains.

To begin the discussion of keyed structures, we define several terms which will be used in the remainder of the paper.

K key space - a set of possible values for the key domain. The key domain (or primary key) is the domain of the relation which is used in determining the storage location for tuples in the relation. Several domains may combined to form a single key. However for this discussion we will assume only one domain is used. The key space will be taken to be the interval (a,b) of the real line since other data encodings may be transformed to this set.

F key distribution - a usually unknown probability distribution function which describes how the keys are distributed over the range (a,b).

N the number of tuples in the relation

$\{K_1,\ldots,K_N\}$ the N keys, $K_i \in K$, present in the relation. This is a sample from F. To simplify notation, we will assume the sample has been ordered so that $K_i \leq K_{i+1}$ for all i.

A address space - a set of integers $\{1,2,\ldots,P\}$, each member of the set representing a secondary storage location (page) capable of storing one or more tuples. These P pages are referred to as the "primary pages". When a primary page becomes full, one or more "overflow pages" are linked to it.

P the number of primary pages

C tuple capacity of a page - the number of tuples that can be accommodated on a single page of secondary storage. (page

size divided by tuple size)

C′ key capacity of a page - the number of keys that can be accommodated on a single page of secondary storage. (page size divided by key domain size)

H key to address function - a mapping from key values to addresses H:K->A.

S parameter set - a set of parameters which are used in the key to address transformation, H.

OCF occupancy factor - a measure of secondary storage space usage. It is defined as the total secondary storage space used (primary plus overflow pages) divided by minimum possible space (the minimum space is N/C).

ACF access factor - average number of data page accesses to reach a tuple. This includes the primary data page access and all overflow page accesses but does not include any accesses required by the key to address transformation.

Figure 2.1 shows a small set of sample keys with some of the above parameters specified. This sample set of keys will be used in chapter 3 to illustrate each of the types of key to address functions considered. The differences between types of keyed structures lie in the definition of the key to address function, H. The remainder of this chapter discusses desirable conditions for the function to meet.

Key Space $0 \leq K < 100$

Address Space $0 \leq A < 10$

$P = 10$

$C = 2$

$N = 20$

$\{K_1,\ldots,K_{20}\} = \{1,3,4,7,9,11,12,15,16,19,$
$23,34,38,47,55,62,70,83,90,98\}$

Figure 2.1 A Sample Set of Data

## 2.2 Cost of Key to Address Computation

The first condition essentially states that the key to address computation should not be so complex (in terms of number of parameters) that it requires secondary memory accesses to fetch parameters during address computation.

Condition 1.

> The function should not introduce additional secondary storage accesses in order to compute an address.

## 2.3 Address Space Partition

Condition 2 requires that pages in secondary storage are used in a uniform manner so that overflow areas are not heavily used. Overflow areas are necessary when more than C tuples are mapped to a single address. To access tuples on an overflow page,

first the primary page (the one determined by H) must be accessed and then the overflow page(s) is accessed. The added accesses necessary to retrieve tuples on overflow pages increases ACF.

Condition 2.

The function should map the given sample of the key space uniformly across the address space.

## 2.4 Order Preservation

Up to this point, we have implied that when a key value is specified, it appears in the qualification of the query in the form KEY = VALUE. Queries may, however, involve ranges on domain values. For example

RETRIEVE (Target-list) WHERE E.SALARY < 10000

Certain key to address functions have the property that there is no correspondence between the order of the keys and the order of the addresses to which they are assigned. These functions are of little value in selecting tuples from a range of key values. For this reason, a third condition often must be imposed on the key to address function.

Condition 3.

The function should be an order preserving function (i.e. if $K_1 < K_2$ then $H(K_1) < H(K_2)$ ).

This condition is important whenever it is expected that queries

will involve qualifications which specify a range on the primary key. In such cases it is important to limit the number of tuples scanned to those in the specified range.

A key to address function which satisfied these three conditions would provide efficient data access for the queries in a relational query language.

# CHAPTER 3

## Key to Address Functions

We know of no storage structure that satisfies all three conditions specified in chapter 2 independent of the data stored. In this chapter, several possible structures will be examined which meet some of the conditions. First, randomizing functions will be treated. This type of structure is useful when order preservation (condition 3) is not important. Next we will discuss simple order preserving functions (which usually satisfy conditions 1 and 3 but not 2). Then we consider directory structures (which usually obey conditions 2 and 3 but not 1). Next we discuss generalized directories (which offer a continuum of possibilities between the previous two types). Finally we will look at the tradeoffs between two types of directories, dynamic and static.

### 3.1 Randomizing Functions

A class of functions which usually meets the first two of these conditions is known as randomizing or hash functions. Here, H is chosen so as to spread the keys randomly across the address space. Figure 3.1 shows our example data from chapter 2 with one possible randomizing function. Randomizing functions have been investigated extensively [MORR68, LUM71a, LUM73, DEUT75] and most of these functions have a very small number of parameters. These

| Page | Keys Assigned to Page |
|---|---|
| 0 | 70 90 |
| 1 | 1 11 |
| 2 | 12 62 |
| 3 | 3 23 \| 83 |
| 4 | 4 34 |
| 5 | 15 55 |
| 6 | 16 |
| 7 | 7 47 |
| 8 | 38 98 |
| 9 | 9 19 |

$$H(k) = k \bmod 10$$

Figure 3.1 Randomizing Function

functions have the advantage that they meet both conditions 1 and 2 for a large class of key distribution, F. An excellent compilation of various randomizing functions is given in [LUM71a]. A tutorial on randomizing functions, including a good set of additional references, is found in [MAUR75].

Randomizing functions provide an excellent response to the needs of queries involving equality on the key domain. For a given key value, the function H will return the address which contains all tuples possessing that key value. For example, if the EMPLOYEE relation were randomized with SALARY as the key domain, then the

query

RETRIEVE (Target-list) WHERE E.SALARY = 10000

would only require an average of ACF accesses to find all qualifying tuples. For the above reasons, we will include randomizing functions in the set of basic storage structures to be used to support data relations. They will be used to implement relations which are accessed on equality of the key domain.

## 3.2 Simple Order Preserving Functions

This class of functions requires only minimal parameters as in the case of randomizing functions, yet also preserves order in the address space. An example from this class of functions is to take the j leftmost bits of the key as the address [RIVE74]. The value of j is chosen in order to give an address space of $2^j$ values. Another simple, order preserving function is one which divides the key range (a,b) into equal size buckets and assigns one of the P address values to each bucket. Here H is defined as

$$H(k) = \lfloor P(k-a)/(b-a) \rfloor$$

where $\lfloor x \rfloor$ denotes the least integer greater than x. This function on the example data is shown in Figure 3.2. Note how the uneven distribution in key space is directly reflected in the poor partition of the address space.

The advantage of these functions is that they satisfy condition 1 and thus do not introduce any significant overhead in computation

| Page | Keys Assigned to Page |
|---|---|
| 0 | 1 3 \| 4 7 9 |
| 1 | 11 12 \| 15 16 19 |
| 2 | 23 |
| 3 | 34 38 |
| 4 | 47 |
| 5 | 55 |
| 6 | 62 |
| 7 | 70 |
| 8 | 83 |
| 9 | 90 98 |

$$H(k) = \lfloor k/10 \rfloor$$

Figure 3.2 Simple Order Preserving Function

of addresses. The problem with all functions of this type is that the distribution in address space is directly dependent on the distribution in key space. So unless there is uniformity in the sample key values, there will be bunching in address space which implies many overflows and/or much wasted primary space. Note that for a highly skewed distribution, a simple order preserving function may assign almost all keys to the same address causing ACF to approach N (i.e. access time can be linear in N). Therefore we conclude that simple order preserving functions should be used only when it can be determined that "reasonable" uniformity exists in the key space.

## 3.3 Directory Structures

A normal directory structure is a function which is constructed such that each page contains the same number of tuples and there are initially no overflow pages used. One such function is

$$S = \{L_i \mid L_i = K_{C*i},\ i=1,N/C\}$$

with

$$H(k) = i \text{ for } L_i \leq k < L_{i+1}\ .$$

Here, the parameters of the function are the low key values on each page of secondary storage. Figure 3.3 shows the sample data using the above directory function. This function satisfies condition 2, however it has N/C parameters which means that for non-trivial values of N, the parameters must be stored in secondary memory (violating condition 1). For large values of N, the parameters themselves need to be located via a key to address function, thus creating the common multilevel directory structure (i.e. ISAM [IBM66]). Each level of the directory adds an additional access to the cost of computing a tuple address. The average access time for a tuple is then the directory access time plus the single data page access (here ACF is 1)

$$LOG_C N/C + 1$$

Despite the cost of directory accesses, this structure is currently widely used when ordering is required. One reason for this choice is that for directories the worst case access time is logarithmic (to a large base) in N whereas simple order preserv-

| Page | Keys Assigned to Page |
|---|---|
| 0 | 1 3 |
| 1 | 4 7 |
| 2 | 9 11 |
| 3 | 12 15 |
| 4 | 16 19 |
| 5 | 23 34 |
| 6 | 38 47 |
| 7 | 55 62 |
| 8 | 70 83 |
| 9 | 90 98 |

$$H(k) = \begin{cases} 0 & \text{for } 0 \leq k < 4 \\ 1 & \text{for } 4 \leq k < 9 \\ 2 & \text{for } 9 \leq k < 12 \\ \vdots & \vdots \\ 9 & \text{for } 90 \leq k < 100 \end{cases}$$

Figure 3.3 A Directory

ing functions may be linear in N.

## 3.4 Generalized Directories

We now combine the two previous ideas into a structure which meets conditions 2 and 3 and has fewer parameters than normal directories. The parameters of a "generalized directory", H, are an ordered set of pairs:

$$S = \{(L_i, A_i) \mid L_i \leq K,\ A_i \leq A,\ L_i < L_{i+1}, A_i < A_{i+1},\ i=1,M\}$$

The key to address mapping, H, is:

$$H(k) = A_i + \lfloor (A_{i+1} - A_i)(k - L_i)/(L_{i+1} - L_i) \rfloor$$
$$\text{for } L_i \leq k < L_{i+1}$$

This type of function divides the key space into M intervals which may be of varying sizes and assigns to the $i^{th}$ interval $A_{i+1} - A_i$ pages of secondary storage. Within an interval a simple order preserving function is used to divide the key range equally into the pages assigned to that region. Functions of this nature have been investigated by [FEHR75] and [WHIT75]. Figure 3.4 indicates one generalized directory for the sample data.

A "data independent directory" is one in which the choice of H is made without any knowledge of the distribution of keys within the interval (a,b). An example of such a directory is the simple order preserving function described above where

$$M = 2$$
$$(L_1, A_1) = (a, 1)$$
$$(L_2, A_2) = (b, P)$$

A "data dependent directory" is one in which the sample $\{K_1, \ldots, K_N\}$ from the unknown distribution, F, of keys and is used during construction of H. One example of a data dependent directory is the normal directory discussed above where the $L_i$ are

| Page | Keys Assigned to Page |
|---|---|
| 0 | 1 3 |
| 1 | 4 7 |
| 2 | 9 11 |
| 3 | 12 15 |
| 4 | 16 19 |
| 5 | 23 34 |
| 6 | 38 47 |
| 7 | 55 62 |
| 8 | 70 83 |
| 9 | 90 98 |

$$H(k) = \begin{cases} \lfloor k/4 \rfloor & \text{for } 0 \leq k < 20 \\ 5 + \lfloor (k-20)/16 \rfloor & \text{for } 20 \leq k < 100 \end{cases}$$

Figure 3.4 Generalized Directory

chosen to be the low keys on each secondary storage page.

$$S = \{(L_i, A_i) \mid L_i = K_{i*C},\ A_i = A_{i-1}+1,\ i=1,N/C\}$$

We define a best general directory to be one which satisfies the following optimization problem.

given a collection $\{K_1,\ldots,K_N\}$ of keys

choose H (as defined above)

with minimum average access time

$LOG_C \cdot M + ACF$

subject to the constraint that the total storage space is less than some factor, $f_1$, greater than the minimum possible storage requirement (N/C pages)

i.e. OCF < $f_1$

The solution to this problem will provide a directory which has the best average access time for the given limitation on total storage space. This optimization attempts to minimize the size, M, of the directory while keeping the overflows to a minimum and remaining within the storage constraints.

Usually the performance of the two limiting cases of generalized directories, simple order preserving functions and pure directories, will not be optimal. In the case of data independent directories, the directory may not be a close approximation of the actual distribution, F, or the initial sample $\{K_1,\ldots,K_N\}$. Therefore, H may map more than C tuples to many addresses requiring the use of excessive overflow pages. On the other hand, normal directories provide an even distribution of keys over address space, however; they require a large number of entries in the directory. Thus average access time in the normal directory may be larger than necessary because of the need to make several accesses to compute the address.

An optimal solution to the above problem would require a prohibitive amount of computation due to the number of degrees of freedom. There are many approaches which could be taken in attempting to solve a simpler problem that approximates the one stated

above. We have chosen to look for an approach which can be implemented by an algorithm which only requires a single pass over the data relation. We therefore redefine the problem as that of finding a minimum directory size (minimum M) for fixed limits on the access factor (ACF) and the occupancy factor (OCF). In this way the inclusion of C′ as a parameter is avoided. Even a best solution to this problem would require many passes over the data file (sample keys), so we now outline an algorithm which provides a solution to the second problem with a single pass over the data. Hopefully, this is a good approximation to the first optimization problem.

In the following description, we will refer to the "step width" of the directory function. By this we mean the size of the interval in key space which is mapped to a single value (page) in address space for a given interval of the function (i.e. for the interval from $L_i$ to $L_{i+1}$ the step width is $(L_{i+1} - L_i)/(A_{i+1} - A_i)$ ).

The algorithm scans the data keys once from lowest key value to highest. At the beginning of the scan, several guesses are made at the step width of the function. As data keys are read, the performance of each of the guesses is measured by computing the access factor and the occupancy factor which would result if that guess were used. When a point is reached in reading data keys where none of the guesses continues to meet the fixed limits, $f_1$ and $f_2$, on occupancy and access factors,

$$OCF < f_1 \quad \text{and} \quad ACF < f_2$$

then the point just before the last guess fails is taken as the next entry in the directory. This guess is taken as the step width between the previous entry and the new one. A new set of guesses is then made and the algorithm repeats as above until the last data key is read.

Some comments on the guesses:

1. As a result of the large difference between I/O speeds and computation speeds there should be enough CPU time available during a scan of the relation to allow a sizable number of guesses (NGUESS) to be made and tested.
2. If the first key to be scanned in the new interval is $K_i$, then one of the guesses is chosen to be $K_{i+C} - K_i$ (i.e. a step width for which the first page is exactly filled). In the worst case, this guess will be chosen and will meet the constraints for one page of data keys, resulting in a normal directory structure.
3. By choosing the guesses to be $(K_{i+C*2}j - K_i)/2^j$ for j=0,NGUESS-1 we choose points logarithmically distant from the current point and thus get approximations to the slope of the function of both a local and global nature.

The following point should also be carefully noted.

As extra space is made available to the algorithm (by increasing the limit on OCF), the algorithm produces a smaller and smaller directory. This is the opposite of what hap-

pens in a normal directory which increases in size as extra data space is provided. (Recall that in a normal directory there is one parameter for each data page. So, if additional data pages are allocated, the directory will grow in size.)

Some Experimental Results

The following several figures illustrate the performance of the algorithm in a series of experiments. Each graph shows the size of the directory produced by the algorithm against varying occupancy factors. In each case, the size of the directory is given as a percent of the minimum possible normal directory size. The occupancy factor is shown as a percentage of additional storage space beyond the minimum possible space for the relation under consideration (i.e. 100(OCF-1) ).

The first experiment compares the performance of the algorithm on two data relations, one uniformly distributed and the other non-uniformly distributed. The first relation contains 10,000 8-digit numbers in the range 0 to 99,999,999 which were produced by a random number generator. The second relation contains 7,100 names of property owners in Alameda County, California. The distribution of numbers is very nearly uniform while the distribution of names, as would be expected, is quite non-uniform. Figure 3.5 shows results for these two relations for ACF close to 1 (no additional access cost) and also for the name data ACF=1.6. For the nearly uniform number data, an order of magnitude reduc-

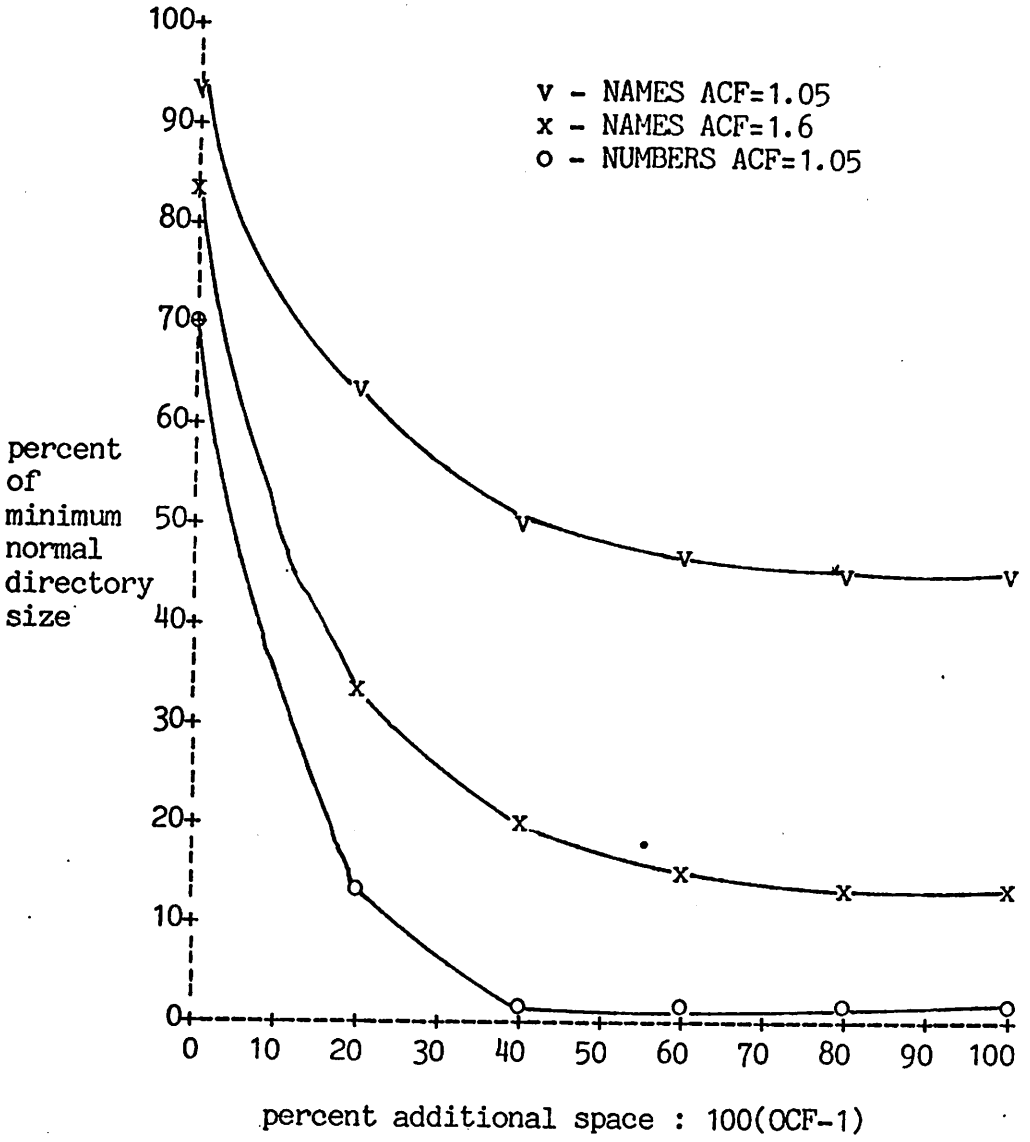


Figure 3.5 Uniform and Non-uniform Data

tion in directory size is achieved at a cost of approximately 22 percent in additional data space and two order of magnitudes reduction in directory size is achieved with about 38 percent added storage. For the same limit on access factor, the reduction in directory size for the non-uniform data is not nearly as great. When ACF is relaxed to 1.6 a two thirds reduction in directory size is realized at a cost of 20 percent in storage

space. For the key size and page size under consideration, if this reduction saves one directory level, then the net savings in access time is 1 - 0.6 = 0.4 accesses per directory search.

The next experiment tests the consistency of the algorithm's performance on two similar sets of data. One set of data is the 7,100 names used previously. The other set of data is 4,760 names of property owners in San Mateo County, California. Figure 3.6 shows that the results of the algorithm are approximately the same for both sets of names.

All of the above experiments were run using NGUESS=10. Figure 3.7 shows the Alameda County name data re-run with 5, 10, and 20 guesses. Although 5 guesses showed somewhat worse performance than 10 guesses, increasing the number of guesses beyond 10 provides almost no improvement in the algorithm's performance. This experiment verified that a small number of guesses (which add little to the running time of the algorithm) provide results which are as good as those achieved with a large number of

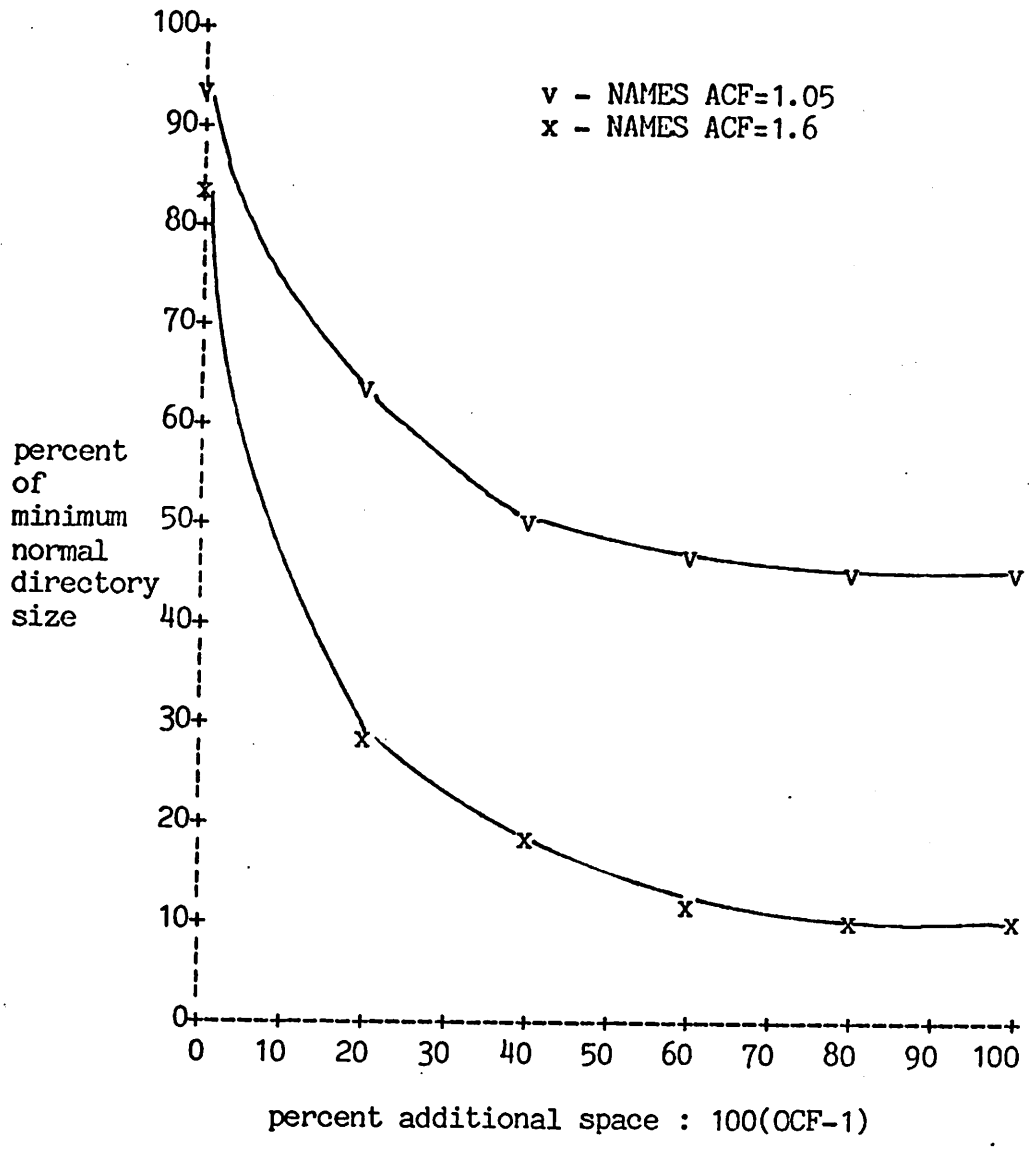


Figure 3.6 Consistency

guesses (which may add substantially to the running time).

The next results indicate the effect of different data page capacities on the algorithm. Again the Alameda County names are used with ACF=1.6. In all of the previous experiments, the page capacity (number of tuples on a data page) was set at 10. Here we use values of 5, 10, 50, and 100 for the page capacity. As seen

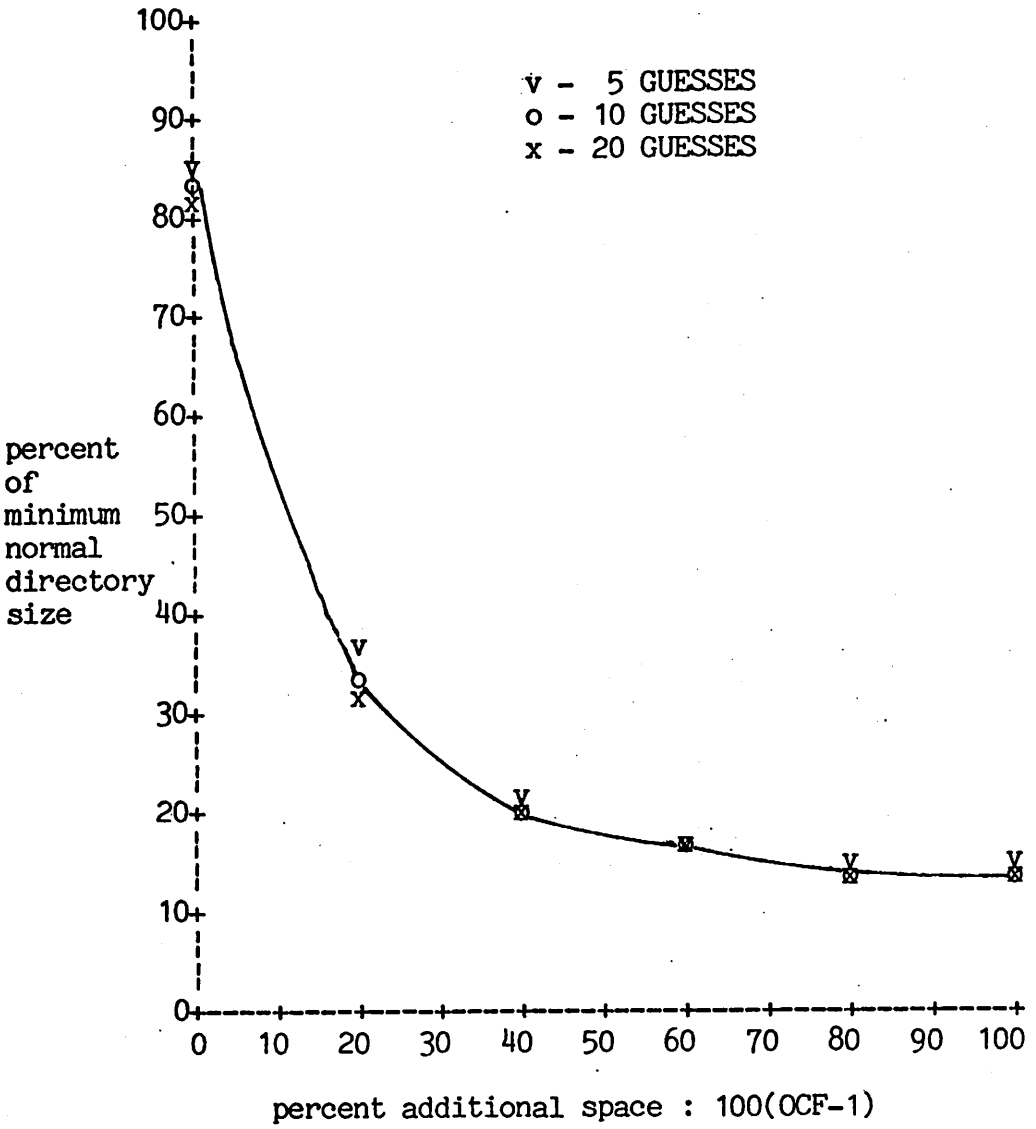


Figure 3.7 Number of Guesses

in Figure 3.8, with a large page capacity (50 or 100 tuples) and with 30 percent or more extra space, the algorithm performed substantially better than for the smaller page capacities. One possible explanation is that by sampling every 50 or 100 data points, the function which is being approximated is a smoother function.

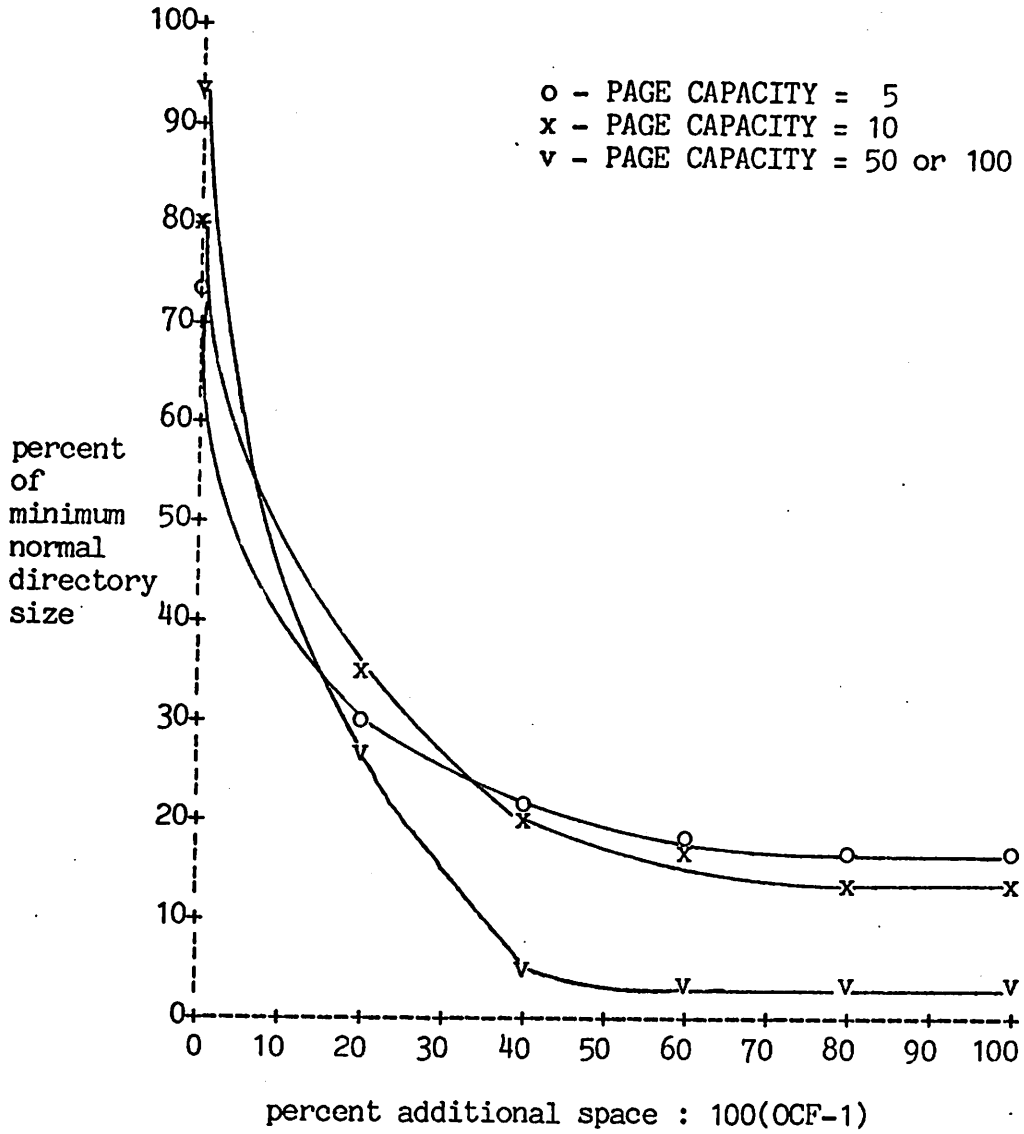


Figure 3.8 Page Capacity

Finally, in figure 3.9, we re-plot the original Alameda County data but instead of using percent of minimum directory size, we use percent of normal directory size for each value of additional space. Thus a normal directory would require 50 percent more parameters than the minimum normal directory if 50 percent added free data space were desired. This graph indicates the improvement which would be gained by using a generalized directory over

a normal directory for different occupancy factors. This graph is very important for two reasons. First, in almost any application where updates are occurring, a normal directory would be built with a certain percentage of free space. Secondly, the algorithm described for generating generalized directories works well when supplied with some amount of free space at directory creation time. These two facts support the contention that even for non-uniform data, generalized directories are preferred over normal directory structures.

The generalized directory provides a structure which complements the randomizing function. We include it in the set of storage structures used to implement relations and will use it whenever queries involving ranges on the primary key are expected. Such a structure will take advantage of whatever uniformity exists in data sets and will never give worse performance than a normal directory.

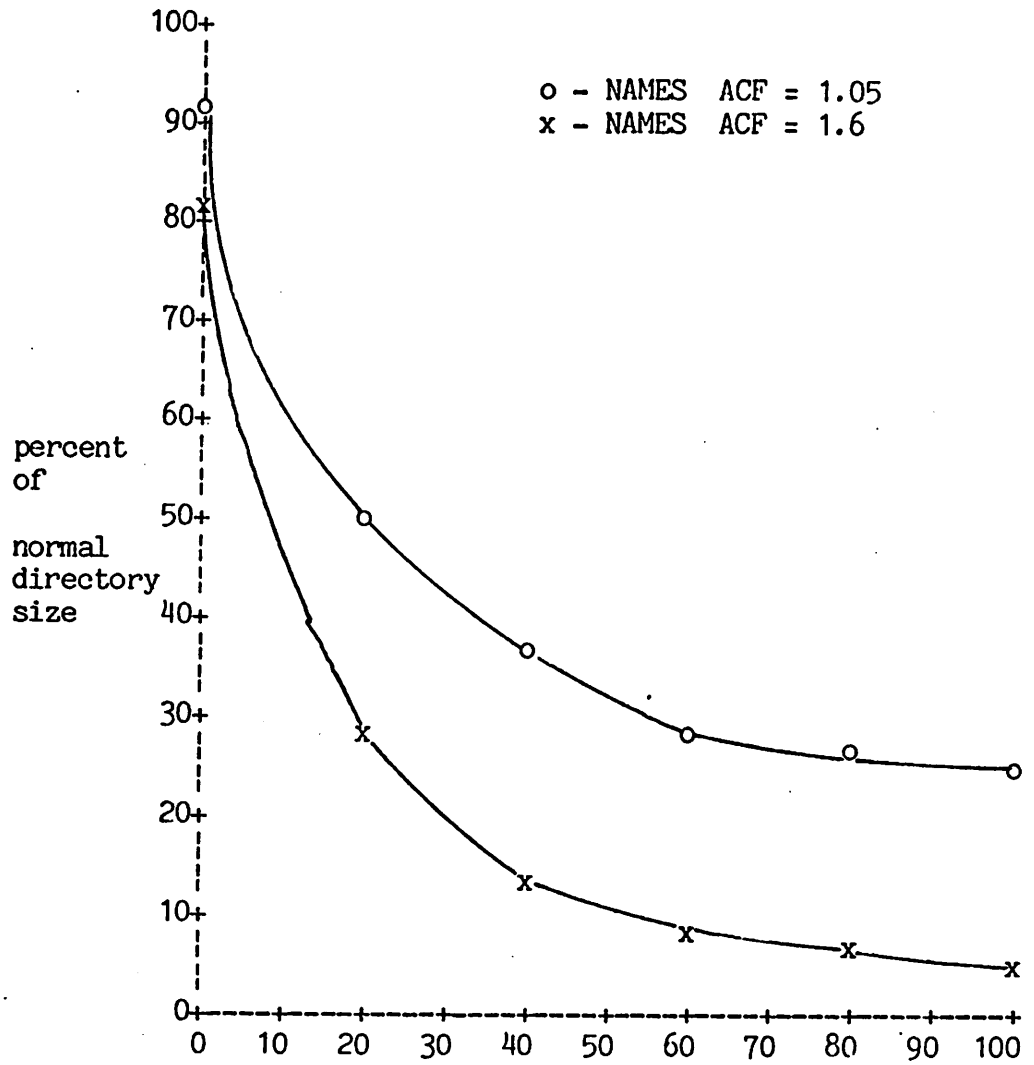


Figure 3.9 Additional Space

## 3.5 Static vs Dynamic Directories

In the above discussion we have only been concerned with the process of building key to address functions and using them for data retrieval. We now consider the problem of choosing a structure

which will be useful in an environment which includes updates to the relation (REPLACE, DELETE, and APPEND). We consider two different approaches to the problem of maintaining directory structures in this environment.

Dynamic Directory Structures

The B-tree [BAYE70] is one example of a class of storage structures which we call a "dynamic directory" because it is dynamically reorganized during updates to provide a balanced search tree (or directory) at all times.

Basically, a B-tree is a balanced tree with between k+1 and 2k+1 sons for any given node. The parameter k is determined by the page size and data characteristics. An example of a B-tree for k=1 is shown in figure 3.10. Here, space exists on each page for two data tuples and three pointers. (In the figure we indicate only the key portion of the tuple.) Note that the tuples are in collating sequence if the tree is scanned in postorder. Note also that the number of page accesses required to reference any given tuple is logarithmic in the number of data tuples.

The major advantage of this structure is that the tree can be kept balanced during insertions and deletions with a known (and small) worst case update cost. Hence, a small worst case search time is always guaranteed.

For example, the tuple with key ALL can be added to the tree with only three page accesses and stored in the empty space on page A.

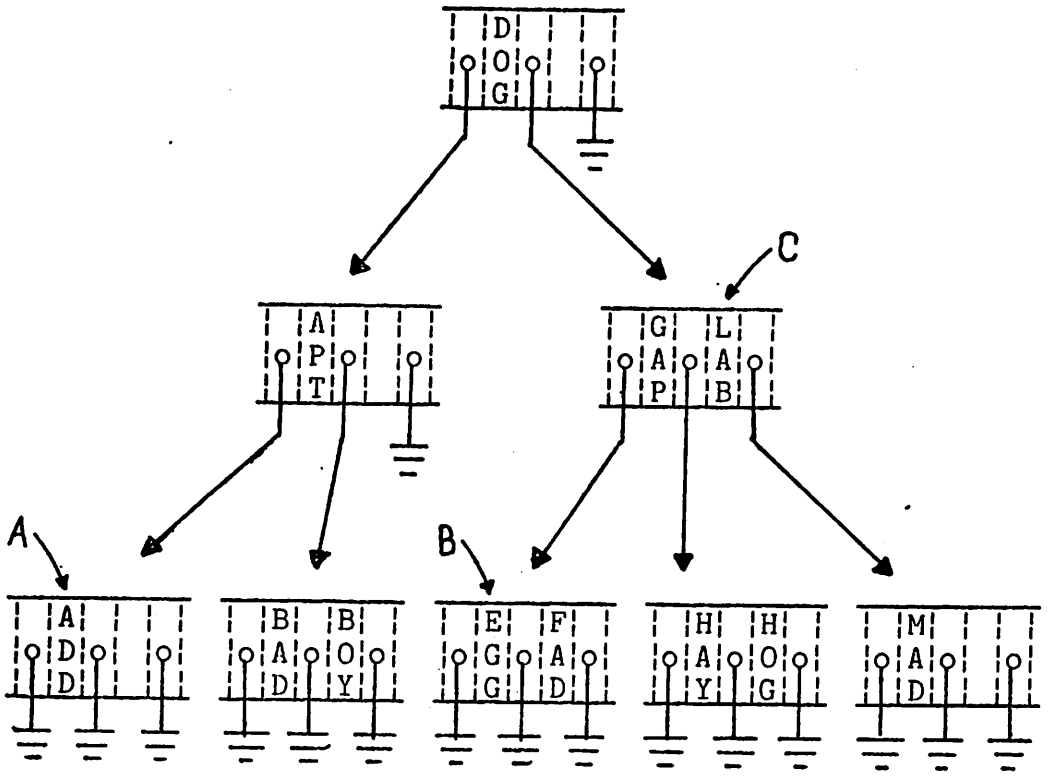


Figure 3.10 A B-Tree

However, because page B is full, insertion of a tuple with key ELF would cause B to be split into two pages, and a tuple to be moved to a higher level node. This in turn causes page C to split as well; the resultant structure is the balanced tree shown in figure 3.11. It is important to note the necessity here of relocating tuples within the relation; such address modifications will later be shown to be potentially troublesome.

There are several variations on the basic dynamic directory theme which offer certain obvious advantages over B-trees [KNUT73, KEEH74]. The original proposal placed entire tuples in directory

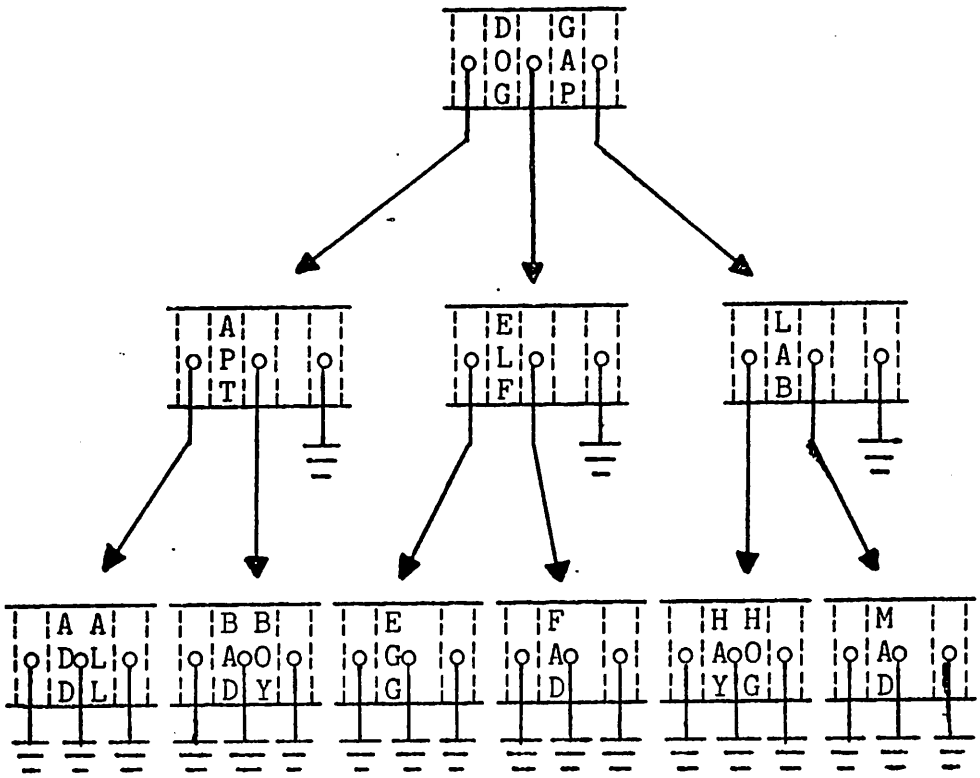


Figure 3.11 The Updated B-Tree

pages. In fact, placing only keys in the directory increases k and reduces the height of the tree. Hence, data pages can be accessed with fewer retrievals from secondary storage. Also, the minimum number of tuples per page can be increased beyond k+1 by employing an overflow strategy onto adjacent pages which reduces the number of page splits. This idea underlies B*-trees [KNUT73]. VSAM [KEEH74] is another variant of dynamic directories. In the sequel we will be concerned solely with dynamic directory structures having only keys in the directory levels.

Static Directory Structures

Storage structures for which index levels are not changed dynamically, such as ISAM [IBM66], will be termed "static directories". Figure 3.12 represents one such structure for the same data used in figure 3.10. Four important features of static directories should be noted:

1) The index levels are formed by recording the high key on each data page.

2) Once formed, the index levels are NOT dynamically altered (in contrast to a dynamic directory).

3) As a consequence of (2), only one pointer per index page is required. In our example, the three data pages pointed to can be logically (or physically) contiguous. Because of this pointer suppression, we have assumed three keys will fit on a static directory page instead of the two in a dynamic directory.

4) Additions to the structure of figure 3.12 are handled by chaining into overflow areas. The addition of a tuple for ALL would cause page A to split, and an overflow page to be allocated and chained onto page A as shown in figure 3.13. Note that existing tuples are not moved and that their ordering within a primary page and its overflow pages is not guaranteed. If tuples must be kept in collating sequence, however, they can easily be chained together. Garbage col-

lection on deleted tuples is also easily accomplished, though the mechanism is not discussed here.

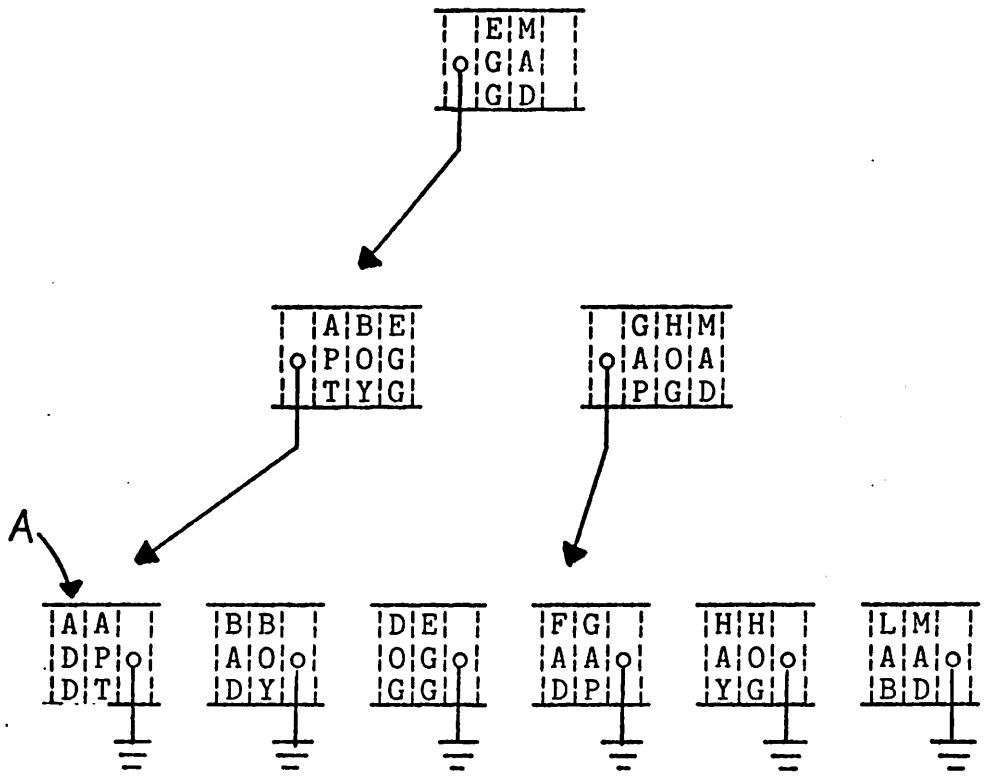


Figure 3.12 A Static Structure

As a storage structure for a single isolated data file, dynamic directories appear very useful, however, in a data base environment we feel there are three important points to be considered before adopting such a structure.

Points of Comparison Between Dynamic and Static Directories

1. Secondary Indices

If it becomes necessary to access a relation on some portion of a


Figure 3.13 A Portion of the Updated Static Structure

tuple that is not a directory key, a complete sequential scan of the relation may be required. In our sample data base, for instance, to access all tuples with keys ending in Y would entail just such a search. This problem is often alleviated, however, by using secondary indices [STON74c].

In the case of secondary indices, an index relation is created which contains pairs of attribute values and pointers to tuples in the primary relation which have that value. Figure 3.14 gives an example of a secondary index that might be used in conjunction with the structure of figure 3.10; ->BAD indicates a pointer to the tuple for BAD in the primary relation.

Independent of the storage structure chosen for the data relation, there is always some amount of overhead in maintaining such indices during updates to the primary relation. When the data

| | |
|---|---|
| B | ->LAB |
| D | ->BAD |
| D | ->ADD |
| D | ->MAD |
| D | ->FAD |
| G | ->HOG |
| G | ->EGG |
| G | ->DOG |
| P | ->GAP |
| T | ->APT |
| Y | ->BOY |
| Y | ->HAY_ |

Figure 3.14 An Inversion on the Last Letter of a Key

relation is updated, so, too must the indices be updated to reflect newly added, deleted, or changed values of the inverted attributes. If the data relation is a dynamic directory, however, secondary updates may also be generated in the course of dynamic reorganization. These additional updates occur whenever a primary update causes data pages to split or merge (as in the second insertion into the B-tree of figure 3.10). In such cases tuples must be moved to new pages and be assigned new logical (or physical) page addresses. Every tuple which is assigned a new address then requires an update for each existing secondary index. Such additional updates could be avoided if the secondary index used the primary key of the data tuple instead of its address as a pointer. This approach, however, entails primary key decoding through the directory for each access using a secondary index.

2. Concurrency

The second problem with dynamic structures arises when concurrent processes use the same relation. Suppose two processes are simultaneously accessing a dynamic directory; one inserting a tuple and the other scanning a portion of the tree. Suppose further that the scanning process is partway through a page when the updating process causes that page to be split by an insertion. This rearrangement will leave the scanning process pointing to a wrong (or non existent) tuple unless the updating process alters the scan pointer in a non-trivial way.

Other problems arise when two processes concurrently update the same B*-tree. Suppose that two processes are adding tuples to adjacent pages in the tree, and that both pages are full. First, each process must lock the page it is updating before it can be altered. Then, each must examine the two adjacent pages to see if tuples can be spilled into them to avoid a page split. Unfortunately, they will find that one of the two adjacent pages has already been locked by the competing process. Clearly, this deadlock condition must be recognized and broken.

3. Directory Height

The third problem with dynamic structures involves the height of the directory tree. Because pages are split on the fly in a dynamic directory, explicit pointers to data pages must be present in the higher levels of a dynamic directory. Notice, for example, that space must be left for three pointers on each page of figure 3.10. These pointers consume space and limit the value

of k that can be attained.

These problems can all be avoided in static directory structures. A static directory can have the property that tuples are never moved if chaining between tuples in the primary and overflow areas is allowed, or if the tuples on a given page and its overflow pages are not kept in collating sequence (figure 3.13). If this is the case, pointers can be safely used in secondary indices. Moreover, the directory itself is static; an updating process need only lock the page it is modifying. Also, since tuples are not moved there is no danger of creating intermediate pointers to non-existent tuples. In addition, directory pointers can easily be suppressed, increasing the fanout possible (often by as much as a factor of two). Frequently, this will save one level in the directory. Also, reduction of directory height in static directories can be accomplished using the generalized directory described previously. However, generalized dynamic directories are not practical since a dynamic change in a segment of a generalized directory may require the movement of many pages full of tuples (the better the generalized directory, the worse the effect of a change would be).

In order to further compare performance, we present a simple example. Suppose pointers and keys are both four bytes in length and that the page size is 512 bytes (this is the page size used in the UNIX operating system [RITC73] on top of which INGRES is implemented). In this case, each node in a B*-tree (assuming

only keys are present in the index levels) has between 32 and 63 sons. On the other hand, the number of sons for an indexed structure similar to figure 3.12 is 127. The following table indicates the height of the tree for various sizes of primary data relations being indexed. In both cases we assume the index nodes are completely full.

| number of data pages | tree height with static directory |
|---|---|
| 2-127 | 2 |
| 128-16,129 | 3 |
| 16,130-2,048,383 | 4 |

| number of data pages | tree height with dynamic directory |
|---|---|
| 2-63 | 2 |
| 64-3,968 | 3 |
| 3,969-250,047 | 4 |
| 250,048-15,752,961 | 5 |

The important point to note from the table is that a static directory saves a level in the tree for any relation between 3969 and 16,129 pages (roughly 2-15 million bytes) and for relations over 125 million bytes. For such relations, a static directory requires one less disk read than a dynamic directory for each retrieval request. Of course, this example is sensitive to the page size, key size and pointer size chosen. However, a level is saved in many situations.

Finally, we look at the reorganization cost of static and dynamic

directories. We have already seen that static directories (especially generalized static directories) very often save a level in directory height when compared to a dynamic directory on the same data. The following analysis considers those cases where the static directory saves one level in directory height and compares the access costs to a dynamic directory. We make the following assumptions:

1. Both relations are initially loaded with P primary data pages each containing $C^* < C$ tuples (where C is the maximum number of tuples possible on a data page).
2. No overflow pages are initially used.
3. The height of the static directory is one less than the height of the dynamic directory.
4. The queries in the period after the relation is loaded consist of R retrievals and I inserts.

Let Q be the probability that a page splits (or overflows) during the course of the I inserts. Here we assume that the probability of a page splitting twice is negligible. Because of the difference in directory height, we know that for retrievals the static directory saves one access if the desired data page has not overflowed and there is no savings if the page has overflowed. For inserts, there are three cases to consider. First, if the page is not split, the static directory saves one level. Also, if the current insert causes a page split, one access is saved. Finally, if the page was already split, the two directories require the same number of accesses.

Now we assume the worst case query makeup for the static directory; all I inserts occur before the R retrievals are processed. The inserts and retrievals are followed by a reorganization of the static directory. The net access savings by the static directory is retrieval savings plus insert savings minus reorganization cost. The retrieval savings is R(1-Q) while the insert savings is the number of page splits, QP, plus the sum of all inserts onto primary data pages. To approximate this last figure, we will assume that when the directory was built, each data page was built with only space for one additional tuple (i.e. $C^* = C-1$). Thus the sum of inserts onto data pages must be at least QP making the total savings from insertions at least 2QP. Finally the cost of reorganizing the static directory is assumed to require access to all primary data pages plus all overflow pages. This cost is P(1+Q) accesses. When the net savings from all of the above accesses is positive

$$R(1-Q) + 2QP - P(1+Q) > 0$$

then the static directory will have saved enough accesses to pay for reorganization. This equation simplifies to $R > P$ for positive values of S (for S=0 no reorganization is required). The results indicate that, under the assumptions stated, static directories can be reorganized at any time after the number of retrievals has reached the number of pages in the original relation and still out perform dynamic directories. Thus under these assumptions (which tend to favor dynamic directories), the result indicates that static directories are preferable in all cases

except those where update frequency is so high that a reorganization is required before P retrievals are requested.

## 3.6 A Set of Storage Structures for Relations

We know of no single storage structure which satisfies all three conditions specified in chapter two. We have therefore looked for a set of structures which can be used in different cases which will satisfy some of the conditions. In cases where order preservation is not important, randomizing functions satisfy the remaining two conditions. In the cases where order preservation is important, we have chosen generalized directories as the structure which most closely satisfies the remaining two conditions. For these directories we have provided a linear algorithm for their creation and have argued that a static directory is to be preferred to one which is dynamically reorganized. We now briefly mention that for either the randomizing or directory structures, data compression techniques [GOTT75] can be used to provide a savings in secondary storage space at a cost of increased computation time for decoding and encoding during retrieval and update. Adding a compressed form for each of the two structures gives us the set of storage structures which are used to implement relations in INGRES:

1. Randomized Structures
2. Generalized Directory Structures
3. Compressed Randomized Strucutres

# 4. Compressed Generalized Directory Structures

# CHAPTER 4

# Auxiliary Structures

Thus far we have considered different choices of storage structures for data relations. These choices provide fast access to tuples if the key domain(s) is specified. In this chapter, we will consider methods of storing additional redundant information which will increase access speed when the primary access key is not specified and some other secondary domains are specified. Also we will consider how redundant data storage can speed the processing of aggregates, aggregate functions, and multi-relation queries. Each of the following sections will examine a potential auxiliary structure. For each structure, references will be given to previous uses of such a structure (if any), an example in QUEL will be given of when the structure would be useful, an indication will be made as to the difficulty of updating the structure, and finally a judgement as to the structures overall value will be made. We begin with the most common form of auxiliary structure, the secondary index.

## 4.1 Secondary Indices on Attributes

A secondary index (inversion) on a domain is a binary relation between values of the domain and tuples (or tuple identifiers) from the data relation. The reason for creating a secondary

index is to provide fast access via the index domain when the data relation key is unavailable. For example, in the query

RETRIEVE E.NAME WHERE E.SALARY = 10000

a full scan of the EMPLOYEE relation would normally be required if the key for that relation were, for instance, NAME. However, if a secondary index existed on SALARY, the number of employee tuples examined could be substantially limited. To create the index requires an operation equivalent to

RETRIEVE INTO SALINDEX(E.SALARY, PTR= E.TID)

Figure 4.1 shows the result of this query when applied to the

salindex relation

| salary | ptr |
|---|---|
| 10000 | 6 |
| 12000 | 1 |
| 14000 | 4 |
| 14000 | 5 |
| 20000 | 2 |
| 31000 | 3 |

Figure 4.1 A Simple Index

EMPLOYEE relation from chapter 1. Now if this relation is organized so that SALARY is the key domain, then when a value for SALARY is supplied, only a small number of tuples need be accessed from the index and the PTR domain provides direct access to the appropriate tuples in the EMPLOYEE relation.

Secondary indices are used in many data base management systems. In TDMS [BLIE67, BLIE68] all domains are indexed, while in most other systems, only a fraction of the domains are indexed. When-

ever values are specified for more than one indexed domain, then list processing techniques may be used to create a list of possibly qualifying tuple id's. This reduces the number of tuples in the data relation which must be examined. Reducing the number of data tuples examined is important in that each tuple examined via an index can be assumed to require an additional page access [ROTH74a]. In systems with a hierarchy of physical storage devices, the indices may be kept in faster memory than the data relation. This justifies increased index processing (like list intersection) in order to reduce the more costly data relation accesses.

A combined index, as described in [LUM70] and [MULL71], is another approach which takes advantage of several domain values being specified. Here, the index relation consists of a projection of several domains and the TID from the data relation instead of just a single domain as in a simple index. If the index relation is ordered on all projected domains from left to right, then specifying values for any left subset of the index domains will limit the number of data tuples examined. The more left domain values specified, the finer will be the resolution on the data relation. Combined indices have the advantage that list processing is not necessary in order to get a fine resolution, however, for k attribute values to be useful in improving resolution, the k attributes must be the leftmost k attributes of the index.

Updating secondary indices is a straightforward operation. If an update statement results in a number of before and after images of tuples (as is the case in INGRES), then an index update must only delete the before value of the attribute from the index and insert the after value. Since the index itself is keyed on the attribute and a value is provided for the attribute in both the delete and insert operations, the update cost is only the cost of 2 key to address transformations. Each APPEND or DELETE query will require all secondary indices to be updated for each tuple added or removed from the data relation. A REPLACE query will require updates to all secondary indices which include domains that were modified in the data relation (all domains named in the target-list of the REPLACE). A combined index on k domains will require about 1/k of the work required to update k single domain indices.

## 4.2 Secondary Indices on Functions of Attributes

A simple generalization of secondary indices allows them to be useful in many cases where the normal secondary index can not be used. The following example illustrates such a case.

```
RETRIEVE E.NAME WHERE SQRT(E.XYZ) = 4
                AND E.START - E.BIRTH = 25
```

Assuming that the data relation, EMPLOYEE, is keyed on NAME, one might attempt to utilize secondary indices on XYZ, BIRTH, or START to limit the scan of EMPLOYEE. However, in the cases of

BIRTH and START, there is no value specified for either of the attributes alone. And in the case of XYZ, it would require taking the inverse of the SQRT function to put the clause into the form E.XYZ = 16 which could then make use of an index on XYZ. In order to handle this type of query efficiently, we generalize secondary indices to allow indices on functions of attributes as well as indices on simple attributes. Now an index could be maintained for values of SQRT(E.XYZ) or for values of E.START-E.BIRTH. Figure 4.2 indicates an attribute function index on

start_age relation

| age | ptr |
|---|---|
| 20 | 4 |
| 20 | 6 |
| 25 | 5 |
| 26 | 1 |
| 28 | 2 |
| 32 | 3 |

Figure 4.2 An Attribute Function Index

E.START-E.BIRTH. Two points must be considered about such indices: 1) recognition of when they can be used, and 2) how they are updated. The recognition problem can be solved to varying degrees of completeness. One approach is to take attribute functions as they appear in the query and look them up in a table of indices for attribute functions. More complete approaches will make greater attempts to find indices which exist for equivalent functions. This may be done by putting clauses into a canonical form. For instance, some arithmetic expression manipulation may

be done to put constants on the right of a relational operator (=,<,>, etc.). Also, attributes may be commuted to order them alphabetically (or by domain number) in the expression. So the clause

$$X.B - 5 + X.A < X.C$$

might have the canonical form

$$X.A + X.B - X.C < 5$$

The number of equivalent expressions which can be recognized will depend on the goodness of the arithmetic expression manipulator. It is believed, however, that only a few forms for an expression will normally be used and so even a very simple expression manipulation algorithm will probably suffice.

The second concern with generalization of secondary indices is in conjunction with updates. If updates to the data relation generate before and after images of tuples for use in secondary index updates (as is done in INGRES), then the updating of general indices is straightforward. Whenever any attribute in the attribute function being indexed is involved in an update, the attribute function must be calculated on both the before and after image of the tuple. If a difference exists, the before value is deleted from the index and the after value is added. Thus the update procedure is almost identical to simple indices and is not a problem.

There are many cases where a function of one or more attributes is frequently specified in queries, but being computable from

other attributes it need not be stored as a new attribute. It is in such cases that the generalization of indices to attribute functions is very important.

## 4.3 Predicates on a Single Relation

We have thus far considered auxiliary structures for simple attributes and attribute functions. The next larger piece of a query is the clause. The clause is a truth valued unit which may form the entire qualification or may be joined to other clauses by boolean operators to make up the qualification. In this section we will discuss clauses which involve only a single variable (one relation). These clauses are predicates which perform a restriction on the relation. An example of such a clause is

E.DEPT = "toy" .

If the same clause appears frequently enough in queries, it would be convenient to be able to quickly identify the restricted subset of the relation which satisfy the clause. This can be done by creating an auxiliary relation which contains TID's of only the restricted set. To create such an auxiliary relation for the example requires the query

RETRIEVE INTO TOY_DEPT(PTR= E.TID) WHERE E.DEPT="toy"

For predicates on single relations, the recognition and update problems are very similar to those for secondary indices on functions of attributes. Actually, the only difference between the two are 1) the size of the auxiliary relation may be considerably

smaller for predicates, but 2) the secondary index may be useful in more queries (i.e. may be used in several clauses). The predicate is in fact a restriction of an attribute function secondary index. In cases where there are a small number of values for the attribute function and/or where a particular value is used very frequently, the predicate relation may be preferred. A more obvious use of predicates is where they involve more than one relation.

## 4.4 Predicates on Multiple Relations

When a clause involves more than one variable (relation), the meaning is that it is a restriction of the cross product space of all relations involved. One use of this type of clause is to provide inter-relation access paths similar to DBTG sets [CODA71a]. A very common example of such a clause is the EQUI-JOIN as typified by the clause

E.MANAGER = M.NAME

which links employees to their respective manager. (the range of M is also EMPLOYEE). The predicate here involves two tuple variables ranging over the same relation; however, in general, the range of these variables may be (and normally is) over two different relations. The useful auxiliary relation here is the relation of all pairs of TID's which satisfy the predicate. To

form the auxiliary relation requires the query

```
RETRIEVE INTO EMP_MGR_LINK(EMPPTR=E.TID,MGRPTR=M.TID)
    WHERE E.MANAGER = M.NAME
```

emp_mgr_link relation

| empptr | mgrptr |
|---|---|
| 1 | 2 |
| 2 | 3 |
| 4 | 3 |
| 5 | 4 |
| 6 | 5 |

Figure 4.3 A Multiple Relation Link

Figure 4.3 indicates the result of this query. In the worst case, this relation may be of size equal to the size of the cross product of the relations involved. However, it is common that the relationship links each tuple in one relation to only one tuple in the other. In such cases, the number of tuples in the link relation is on the order of the number of tuples in one of the data relations. It is in these cases that the existence of a multi-relation link can greatly reduce the number of tuples examined in a search. The use of this type of links in relational systems was first suggested in [TSIC75].

Again one must return to the two questions of recognition and update. As in the case of attribute functions and single relation predicates, the recognition problem is one of attempting to put the clause into a semi-canonical form and then to do a simple look up to see if an auxiliary structure exists. The update problem, however, is considerably more difficult. The problem is

that each relation which is involved in the multi-relation predicate is updated individually. To understand the amount of work required to update a multi-relation predicate, consider the case of a predicate, Q(X,Y), over two relations X and Y where the two relations are the same size (N tuples) and the attributes involved in the predicate are not themselves indexed. A more complete treatment of links which considers combinations of links and secondary indices will appear in [STON75b]. In the case under consideration, if a single tuple, x, is added (APPENDed) to relation X, then all N tuples in relation Y must be tested to see if they satisfy the predicate Q(x,Y). For each tuple in Y which satisfies the predicate, a new tuple is added to the link relation. Deletions from either relation require a scan of the link relation to remove all tuples with the deleted TID. This operation can be made fast for one of the relations by making the pointer to that relation the key of the link relation. Modification of a tuple in either relation (using REPLACE) requires essentially the work of a DELETE followed by an APPEND. Thus we see that updating multi-relation links can be very expensive.

Multi-relation predicates may still be useful in a non-updated fashion. The additional cost is small to create the auxiliary relation during the processing of a query involving such a predicate. If subsequent queries involve one or more uses of the auxiliary structure before an update occurs, then the auxiliary structure will have been worthwhile. When an update occurs to any relation involved in the multi-relation predicate, then the

auxiliary structure is discarded. This type of structure will be useful in applications where a complicated access path is used frequently for retrievals, and updates occur infrequently or periodically.

## 4.5 Aggregation

Another part of a query which lends itself to an auxiliary structure is aggregation. In many applications, it will be common for certain aggregates and aggregate functions to be re-used frequently. A single auxiliary relation can be maintained which contains all aggregate values that have been calculated. The recognition problem is again similar to that mentioned above. The update problem for aggregates, as was the case for multi-relation predicates, is prohibitively expensive. The cost, however, is very small to maintain non-updated aggregate values. All that is required is a single auxiliary relation like AGGVALUES(RELID, AGGTOKENS, AGGVAL, AGGTIME) where RELID specifies the relation over which the aggregation has been performed, AGGTOKENS is the canonical form of the aggregation, AGGVAL is the value computed, and AGGTIME is the time at which the aggregation occurred. When an aggregation appears in a query, it is first looked up in the AGGVALUES relation (where RELID and AGGTOKENS are the primary access key). If it is found and the AGGTIME is more recent than the last modification time to the data relation, then the value, AGGVAL, is used. Otherwise a new value for the

aggregate is calculated and entered into the AGGVALUES relation after being used in the query. A similar approach can be used for aggregation over several relations and for aggregate functions. In the case of aggregate functions, the aggregate function is maintained as another auxiliary relation and the AGGVALUES relation would contain the name of the aggregate function relation instead of a simple AGGVAL.

## 4.6 A Set of Auxiliary Information Structures

In this chapter we have presented several possible types of auxiliary structures which might be used to speed access to data relations. We have considered structures which correspond to increasingly larger portions of a query from the simple attribute through the multi-relation predicate and aggregate functions. The structures which have been found useful are:

1. Secondary indices on simple attributes
2. Secondary indices on attribute functions
3. Non-updated aggregate values and aggregate functions

Multi-relation predicates were found to be of some value in certain retrieve-only situations but they will not be considered in the remainder of the discussion. The next chapter will take the set of storage structures from chapter three and the set of auxiliary structures from this chapter and provide a strategy for determining which structures should be used for a particular data relation.

# CHAPTER 5

## Storage Structure Selection Strategy

We have described previously the set of storage structures which will be used to implement data relations and a set of auxiliary structures used to speed access to data. Now we consider the problem of making specific choices from this set of alternatives for a particular data base. First, in section 5.1, a general strategy for storage structure choices is described. This strategy includes dynamic adaption to the local demands of a single query and periodic response to statistics covering a sequence of many interactions. Almost all previous work in the area of storage structure selection has been done on selection of secondary indices. In section 5.2, this work is reviewed and in 5.3, we define a more widely applicable model for solving the secondary key selection problem. Then in section 5.4, a method for obtaining each of the parameters in the model is described. The model is extended, in section 5.5, to handle attribute function indices, primary key selection, and data relation storage structure. Next, we look at the computational cost of the selection process and indicate that heuristics must be used for large problems. The desirable characteristics of such an algorithm are described and one possible algorithm is presented. Finally, in section 5.7, we suggest a method of monitoring the performance of the choices made.

## 5.1 Dynamic and Periodic Decisions

Our overall strategy recognizes two environments in which storage structure decisions must be made. The first situation occurs during the processing of queries (decomposition) and choices here will be referred to as dynamic decisions. These are decisions for storage structure changes which affect the processing of the current query and which must be carried out immediately to have an effect.

An example of a dynamic decision is the building of a secondary index on an attribute in order to speed the processing of a multi-relation query. For instance, in the query

```
RANGE OF E IS EMPLOYEE
RANGE OF D IS DEPARTMENT
RETRIEVE (E.NAME, D.LOCATION)
        WHERE E.DEPT = D.NAME
```

we require all pairs of employees and departments where the employee's department (DEPT) is the same as the name in the department tuple. If E.DEPT and D.NAME are not key attributes then all pairs must be examined. If, however, an index is dynamically created for one of these attributes, say D.NAME, then for each employee tuple E.DEPT can be used as a key value to the D.NAME index. The non-indexed search requires a complete scan of the department relation for each employee tuple, whereas the indexed method requires only a single index search for each employee tuple. Another example of a dynamic decision is the dis-

carding of a secondary index instead of updating it during the update of a data relation.

Dynamic decisions are an important part of the total storage structure selection strategy. However, each of these decisions is closely tied to the query processing strategy (decomposition) and for that reason we leave this type of decision to be considered as part of the decomposition problem [WONG75].

The second type of storage structure choice is made during periods of low system usage and is called a periodic decision. This type of decision adjusts storage structures to the needs of the average query as opposed to any specific query. An example of such a decision is to change the storage structure of a relation from randomized to sorted or to create a new secondary index on a frequently referenced attribute. In making periodic decisions, the actual problem to be solved is:

> Given a database with many relations, select a storage structure for each relation and a set of auxiliary structures which optimize future query/update performance.

What we require is a process for selecting data relation storage structure, primary key domain, and secondary indices (both simple and attribute function indices). The remainder of this chapter deals with a solution to this problem. The approach used is to start with the selection of secondary indices and then to extend the process to handle the remainder of the problem. First, previous work on secondary index selection is reviewed.

## 5.2 The Key Selection Problem

The problem of secondary index selection has been studied by King [KING74], Stonebraker [STON72, STON74c] and Schkolnick [SCHK74, SCHK75]. In each paper, a model of queries and storage structures is presented, a cost function for comparing indices is given, and finally one or more theorems are described which limit the number of choices to be tested in finding an optimal set of indices. Even with the most complete set of theorems [SCHK75], the optimization problem requires nearly complete enumeration to solve. We will therefore not concentrate on reducing the size of the optimization problem, but on providing a model and cost function which more accurately reflect the data base environment. For now we assume that the optimization problem is to be solved by complete enumeration and we return to the cost of solving the optimization problem in section 5.6.

King uses a simple query model in which each qualification may specify the value of only a single attribute. Stonebraker and Schkolnick allow any number of attributes to be specified but make the strong assumption that specification of any attribute is independent of the specification of any other attribute. The models of Stonebraker and Schkolnick are almost identical. Here, we describe the model as presented in [SCHK75]. The complete set of assumptions that is used is as follows:

1. Queries may involve only one relation.
2. Queries may specify the values of any of the attributes but

the specification is only for equality to the value.

3. Attribute i is specified with probability $p_i$ and is independent of the probability of any other attribute being specified (i.e. $p_{ij} = p_i p_j$, $p_{ijk} = p_i p_j p_k$, etc.).
4. Attribute i has $1/d_i$ distinct values present in the relation and each value is equally likely to be specified.
5. The probability that a query is an update is U. Updates change the value of a single domain of a single tuple and each attribute is equally likely to be updated.

Along with these assumptions, a description is given of the steps involved in processing a query and their associated costs. Given a set, D, of attributes which are indexed, the average cost for processing a query is calculated as follows. (Note that some of the equations have been slightly modified in order to conform to our cost criterion of page accesses as opposed to the seek and transfer time used in [SCHK75].

1. List Formation - The first step in processing a query is to form the lists of tuple identifiers (TIDs) for any attribute which is specified in the query and is in the index set. This cost is proportional to the length of the TID list for the specified value of the attribute. This length is, on the average, $Nd_i$ where $d_i$ is the reciprocal of the number of distinct values for domain i. Thus, if $C'$ TIDs are retrieved on each access to an index page, then the average number of accesses to form a list is $Nd_i/C'$ and for all indices in the set D, the expected cost is

$\sum_{i \in D} p_i N d_i / C'$

2. List Intersection - If more than one list is available, list intersection is performed to create a list of TIDs to be retrieved from the data relation. The cost of this step is considered negligible since it is done in main memory.
3. Tuple Access - All TID's on the resulting list are used to access tuples in the data relation. As a result of assumptions 3 and 4, each list restricts the number of tuples to be accessed by $p_i d_i + (1-p_i)$. So the expected number of tuples retrieved on any query is

   $N \prod_{i \in D} (p_i d_i + 1 - p_i)$ .
4. Tuple Processing - As tuples are retrieved they are tested against the complete qualification and further processing (printing, etc.) takes place. (Tuples retrieved which do not qualify are commonly referred to as "false drops"). This cost is independent of the index set chosen and is therefore not considered.
5. Update Cost - If the query was an update, the appropriate members of the index set must be updated. In [SCHK75] updates are considered to be separate queries which modify only a single data tuple. Since the m attributes are equally likely to be updated, the cost is given as

   $(1/m) \sum_{i \in D} N d_i / C'$

The total expected cost of a query for the set D of indices is

(1-U)(query cost) + U(update cost)

which is

$$(1-U)\Big(\sum_{i\in D} p_i Nd_i/C' + N\prod_{i\in D}(p_i d_i + 1 - p_i)\Big) + U\Big(\sum_{i\in D} Nd_i/C'\Big)$$

## 5.3 An Improved Model for Key Selection

The independence assumption made above leads to a simple analysis; however, it is not an accurate model of the many cases where attributes are highly correlated (positively or negatively) in their appearance in a qualification. We choose a model which allows such dependence between attributes to affect key selection. The set of assumptions stated above is used with the following two important changes:

3*. Attribute i is specified with probability $p_i$. Attributes i and j are specified together in the same query with probability $p_{ij}$ (not necessarily equal to $p_i p_j$). And $p_{ijk}=0$ for all i,j,k; that is, the probability of any three attributes being specified together is negligible.

5*. Attribute i requires an average update cost of $u_i$ for each query. That is, $u_i$ equals the total cost of maintaining an index on attribute i for the set of queries under consideration divided by the number of queries.

This model allows important first order correlations between attributes to be reflected in index selection. The following

analysis could straightforwardly be extended to include second order correlations by using values for $p_{ijk}$ and letting $p_{ijkl}=0$, however, this makes the formulas more difficult to follow.

A second important part of the query model is the manner in which updates are treated. All QUEL-type updates (see Appendix A) are assumed to be allowed and a simple and accurate measure of update cost for each attribute is used. A description of how values for $u_i$ are calculated will be given later.

We now form our cost function based on the processing steps described earlier.

1. List Formation - Recall that the average cost of forming a list for attribute i is $Nd_i/C'$. Instead of forming the expected cost of all such lists, the list formation cost will be added in step 3.
2. List Intersection - Again this step is considered negligible in cost.
3. Tuple Access - Since we have assumed $p_{ijk}=0$ for all i,j,k then the expected cost of accessing tuples can be broken down into 3 cases. Included in each case is the appropriate cost of list formation from step 1.
   i) Two attributes from the index set are specified. Here the cost of the two list formations is added to the cost of tuple access. The assumption made here (as in previous papers) is that the selectivity of the two attributes taken together is the product of their indi-

vidual selectivities (i.e. $d_{ij} = d_i d_j$). So for this case, the expected cost of list formation and tuple access is

$$N\Big( \sum_{i,j \in D} p_{ij}(d_i d_j + (d_i + d_j)/C') \Big) \ .$$

ii) Exactly one attribute from the index set is specified. In this case the cost of one list formation is added to the cost of tuple access yielding an expected cost of

$$N\Big( \sum_{i \in D} p_{i\_only} (d_i + d_i/C') \Big)$$

where $p_{i\_only} = p_i - \sum_{j \in D} p_{ij}$ .

iii) No attribute from the index set is specified. Here, although all N tuples must be accessed, in a sequential scan of the relation, pages of C tuples at a time may be accessed. Hence, N/C accesses must be made for a complete scan and the expected cost for this case is

$$(N/C)\Big( 1 - \sum_{i \in D} p_{i\_only} - \sum_{i,j \in D} p_{ij} \Big) \ .$$

4. Tuple Processing - Again this cost is not considered.

5. Update Cost - If the query was an update, the appropriate members of the index set must be updated. The average cost per query of such updates is $u_i$, so for all indices the cost is

$$\sum_{i \in D} u_i \ .$$

Adding all of the costs, the expected cost of a query for the set D of indices is

$$N\Big( \sum_{i,j \in D} p_{ij}(d_i d_j + (d_i + d_j)/C') + \sum_{i \in D} p_{i\_only}\,(d_i + d_i/C') + (1 - \sum_{i \in D} p_{i\_only} - \sum_{i,j \in D} p_{ij})/C \Big) + \sum_{i \in D} u_i$$

The following example illustrates one important type of situation where the above cost function is superior to previous cost functions ([SCHK75] in particular). The example is for a relation which has 3 attributes, the first of which is very frequent in appearance and is independent of the other two. The remaining two attributes are mutually exclusive in their appearance. We assume the selectivity of each attribute is the same (i.e. $d_1=d_2=d_3$). We also constrain the number of indices allowed to be two. The parameters of interest are

$p_1 = .9 \quad p_2 = .5 \quad p_3 = .5$

$p_{12} = .45 \quad p_{23} = 0$

$p_{13} = .45$

$d_1 = d_2 = d_3 = .01$

$N = 10{,}000$

$C = 1 \quad C' = 10$

$u_1 = u_2 = u_3 = 0$

Below are shown the costs as calculated for each of the three possible pairs of indices. The costs are given in expected number of accesses to process a query. The first cost function, [SCHK75], is labelled COST1 and the new cost function is in the

column labelled COST2. As we see from the tabulation, using COST1, attribute 1 would be selected along with one of the other attributes. However, COST2 takes into consideration the fact that by choosing attributes 2 and 3, all queries will have at least one index available for use and thus will not require any complete scans of the data relation.

| D | COST1 | COST2 |
|---|---|---|
| {1,2} | 550 | 550 |
| {1,3} | 550 | 550 |
| {2,3} | 2250 | 100 |

The example illustrates the importance of including joint probabilities in the cost function. Another common example of a case in which COST2 will make a better choice than COST1 is when two attributes are very likely to appear together.

## 5.4 Obtaining Parameters

Defining a model and cost function is a fruitless exercise if it is difficult or impossible to obtain values for parameters of the model. In this section, we describe a set of relations which may be used to maintain statistical information on interaction conditions and we indicate the queries necessary to obtain values for each of the parameters used in the above cost function. We begin by describing the three statistics relations.

QUALSTATS (QRYID, RELID, ATTID)

This relation provides a record of attributes which appear res-

trictively in a qualification. Each query is assigned a unique identifier, QRYID. For every attribute that restricts the qualification of the query, an entry is made in QUALSTATS with the attribute identifier (ATTID) and the relation to which that attribute belongs (RELID). In this way, QUALSTATS maintains attribute usage information for all attributes in all relations of a database.

UPDATESTATS (QRYID, QRYMODE, RESRELID, NUMUPS)

This relation records the number of tuples updated (NUMUPS) in the result relation (RESRELID) during an APPEND, DELETE or REPLACE query. Again, QRYID is a unique query identifier and QRYMODE indicates the type of update (APPEND, DELETE, or REPLACE).

REPLACESTATS (QRYID, ATTID)

This relation records which attributes are affected during a REPLACE query. Recall that APPEND and DELETE affect entire tuples while REPLACE may modify one or several attributes of a tuple. This relation contains an entry for each attribute which is updated in a REPLACE query.

To obtain the frequency, $p_i$, that each attribute of a relation appears restrictively in a qualification, the following aggregate function is used

```
RANGE OF Q IS QUALSTATS
COUNT(Q.QRYID BY Q.ATTID WHERE Q.RELID="relname")/NUMQRY
```

The aggregate function produces a table of frequencies with an

entry for each attribute (ATTID). Here NUMQRY is the total number of queries in which the relation ("relname") was involved.

```
NUMQRY = COUNT UNIQUE(Q.QRYID WHERE Q.RELID="relname")
```

The joint frequencies, $p_{ij}$, are obtained with

```
RANGE OF Q1 IS QUALSTATS
RANGE OF Q2 IS QUALSTATS
COUNT(Q1.QRYID BY Q1.ATTID,Q2.ATTID
        WHERE Q1.QRYID=Q2.QRYID
        AND   Q1.RELID=Q2.RELID="relname"
    )/NUMQRY
```

The calculation of update costs for each attribute, $u_i$, involves two parts. Recall that $u_i$ is the cost that would be incurred to update an index on attribute i if that index existed. The first part of the cost comes from additions and deletions. Each of these updates requires a corresponding addition or deletion from each index in the index set. The second cost is for tuple modification (REPLACE) which requires an old value to be removed from the appropriate index and a new value to be added (2 operations). Thus the proper calculation of the $u_i$'s is

```
RANGE OF U IS UPDATESTATS
RANGE OF R IS REPLACESTATS
RANGE OF Q IS QRYSTATS

(SUM(U.NUMUPS WHERE U.QRYMODE ≠ "replace"
                AND U.RESRELID = "relname"
    )
```

```
+
SUM(U.NUMUPS*2 BY Q.ATTID WHERE U.QRYMODE = "replace"
                          AND U.QRYID = R.QRYID
                          AND U.RESRELID = "relname"
                          AND R.ATTID = Q.ATTID
    )
)/NUMQRY
```

Finally, the number of unique values for each attribute, $1/d_i$, can be found by using COUNT UNIQUE for each attribute or more likely an estimate of $N/d_i$ can be maintained using sampling techniques.

Among the assumptions stated above, we assumed that the queries are only over a single relation. Although the selection process does not take into account the processing strategy for multi-relation queries, the process of obtaining parameters just described works as well on multi-relation queries. Therefore we can gather statistics from general QUEL queries, even though the use of these parameters in the model will only optimize the selection of keys on one relation at a time.

## 5.5 Extensions to the Selection Process

In chapters 3 and 4, a set of storage structures for data relations and a set of auxiliary structures were presented. The selection process just described is now extended to make the

necessary selections from both of these sets.

Selection of Auxiliary Structures

The selection process just presented chooses attributes for simple secondary indices. The other auxiliary structures presented in chapter 4 which must be handled are aggregate values and indices on attribute functions. In chapter 4 we described a method of maintaining aggregate values. Now we treat the selection of attribute function indices.

Extending the selection process to include secondary indices on attribute functions is essentially a matter of additional bookkeeping. An attribute function can be viewed as a function which creates a new attribute from an old one (i.e. it adds a new column to the relation). Thus, each attribute function can be treated as a new simple attribute with its own values for $p_i$ and $p_{ij}$. A relation similar to QUALSTATS can be used to gather the necessary statistics. Instead of storing an attribute identifier (ATTID), the query processor stores the canonical form of the attribute function. The major difference between an attribute function and a simple attribute is that a separate value for $u_i$ need not be maintained. Since updates to an attribute function index are required at exactly the same times as updates to the attribute used in the function, the value of u for the simple attribute is identical to the value of u for the attribute function.

## Selection of Primary Storage Structures

We turn now to the problem of selecting from the storage structures for data relations considered in chapter 3. First we consider selecting which of the domains is to be the primary key, then we add to the selection process the choice of primary structure type (randomized or directory).

In order to extend the selection procedure to include selection of the primary key, the following modifications are required. First, the problem is restated as: choose the pair (D,k) which minimizes the cost function where D is a set of secondary keys and k is the primary key. If the primary key is specified in a qualification, we assume that it will be used and no secondary indices will be used. Now the cost function must be modified as follows:

1. For each attribute $i \neq k$, set $p_i = p_i - p_{ik}$ and set $p_{ik} = 0$ to reflect the use of the primary key as just described.
2. For the tuple access part of the cost function, introduce a fourth case:

   iv) Primary key specified. In this case, no list formation is done. Also, since the key to address function will group equal key-valued tuples together, C tuples are retrieved with each page access. Thus the cost of primary key access is $Np_k d_k / C$ .
3. The cost of updates to the primary key must be reflected in the update cost. Recall that the update cost is the addi-

tional cost required to update auxiliary structures when an update to the data relation occurs. APPEND and DELETE queries do not cause any additional updates. However a modification of the primary key (by REPLACE) requires the tuple to be relocated (by the key to address function) and thus each secondary index must be updated to reflect the change. In order to compute this cost, recall that the update cost for an attribute, $u_i$, was obtained in two parts; the APPEND, DELETE cost plus the REPLACE cost. If we record these intermediate parameters as $u^1$ (APPEND, DELETE) and $u^2$ (REPLACE) then $u_i = u_i^1 + u_i^2$ and the total update cost becomes

$$\sum_{i \in D} (u_i^1 + u_i^2) + u_k^2 |D|$$

At this point, we have a process for selecting a primary key, k, and a set, D, of secondary keys. We are still working under the assumption that only equality specifications are used in queries. Using this assumption randomized structures are always selected as the storage structure since they provide faster key to address transformation than directories. Assumption 2 above is now relaxed to be:

2*. Queries may specify attributes to be equal to a value or to be within a range of values.

This extension requires a modification to the method of obtaining parameters. As before, $p_i$ is the probability that attribute i is specified (on equality) in the qualification and $d_i$ is the selectivity of that specification. Now, let $r_i$ be the probability that attribute i is specified by a range in the qualification and

let $d_i'$ be the selectivity of the average range specification for attribute i. Every time an attribute is specified to be within a range of values, the selectivity of that range must be estimated. One approach to such an estimate is to assume a uniform distribution of values for the attribute and then simply take the size of the range divided by the maximum range for the attribute (MAX(X.A)-MIN(X.A)) as the selectivity.

Returning to the choice between randomized and directory structures, we saw in chapter 3 the difference between the two structures lies in how the key is specified. For equality specifications, randomized structures provide access to data in approximately one access while directories require several additional accesses to traverse the directory (usually between 2 and 5 additional accesses). However, when a range is specified for the key, randomizing functions require an entire scan of the relation (N/C accesses) while a directory requires a number of accesses proportional to the size of the range. Given that queries on a range of the primary key appear with frequency $r_k$ and that the average size of the range specified is $d_k'$, the expected cost for case (iv) using a directory is

$$r_k(t+d_k'N/C) + p_k(t+d_kN/C) \qquad \text{(iv.a)}$$

where t is the additional number of accesses required for the key to address transformation in the directory (values of t almost always being in the range 2 to 5). If a randomizing stucture is used, the expected cost is

$$r_k(N/C) + p_k(d_kN/C) \qquad \text{(iv.b)}$$

Comparing the two equations, we see that if no range queries are used ($r_k$=0) then a randomized structure is always preferable. However, if range queries are specified at all ($r_k$>0) then a directory is generally preferable. For example, with

$p_k = .50$ $\quad r_k = .01$

$d_k = .001$ $\quad d_k' = .01$

$N = 10{,}000$ $\quad C = 10$

$t = 3$

the cost (iv.a) for a directory is 2.13 accesses while for a randomizing function the cost (iv.b) is 10.5 accesses.

The selection process can now choose the primary key, primary structure type, and secondary indices all at once by finding the triple (D,k,TYPE) which minimizes the modified cost function. Here TYPE indicates either a randomized or directory primary storage structure type. If TYPE indicates a directory then cost (iv.a) is used and if TYPE indicates a randomized structure then cost (iv.b) is used. Thus, for the extended selection process, the cost function is

$$N\Big( \sum_{i,j \in D} p_{ij}(d_i d_j + (d_i + d_j)/C') + \sum_{i \in D} p_{i\_only}\,(d_i + d_i/C') + (1 - \sum_{i \in D} p_{i\_only} - \sum_{i,j \in D} p_{ij} - r_k - p_k)/C \Big) + \text{primary(TYPE)} + \sum_{i \in D} (u_i^1 + u_i^2) + u_k^2 |D|$$

where primary(TYPE) is either (iv.a) or (iv.b) depending on the value of TYPE.

A similar extension can be made for choosing the storage structure (randomized or directory) for each secondary index. However, as shown in [ROTH74a], it is very unlikely that secondary indices involving ranges are useful. We will not therefore give the revised cost function for allowing range specifications through secondary indices. It is however a straightforward exercise to introduce $r_i$ and $d_i'$ into the cost function for secondary indices.

## 5.6 Cost of an Optimal Solution

To this point, we have presented a method for selecting secondary keys and have indicated extensions necessary to choose among the storage structure alternatives that derive from the adoption of primary and auxiliary structures of chapters 3 and 4. What remains to be discussed is the cost of this decision process. To begin the discussion, we consider the secondary key selection method without extensions. If a relation has m domains, then an optimal set of secondary indices can be found by calculating the cost function for each of the $2^m$ possible sets of indices and selecting the one with the smallest cost. In many cases the number of sets to be tested can be substantially reduced by taking note of the following two facts. First, there normally exists a subset of the domains which are never specified restric-

tively in the qualification (i.e. $p_i=0$). These domains need not be considered in key selection process since they would never be chosen. Secondly, there is very often a constraint placed on the number of indices allowed for any relation. This further reduces the subsets of domains to those of size less than the limit. Thus, if we let m* be the number of domains which have $p_i>0$ and L be the limit on the number of indices for a relation, then the number of sets which must be examined to find an optimal solution is $\sum_{i=1,L} \binom{m^*}{i}$. The following table shows the number of sets which must be tested for a variety of values of m* and L. The entries indicate the order of magnitude (power of 10) of the set size.

| | | L | | | | | |
|---|---|---|---|---|---|---|---|
| | | 3 | 5 | 10 | 20 | 50 | 100 |
| m* | 3 | 1 | | | | | |
| | 5 | 1 | 1 | | | | |
| | 10 | 2 | 3 | 3 | | | |
| | 20 | 3 | 4 | 6 | 6 | | |
| | 50 | 4 | 6 | 10 | 13 | 14 | |
| | 100 | 5 | 8 | 13 | 21 | 29 | 29 |

We expect that for many applications, typical values of m* and L will be such that the number of index sets tested will be reasonable. For instance, if computation of the cost function for a single choice of D requires processing time on the order of a millisecond, then for m*=20 and L=5 the time to compute an optimal set is on the order of tens of seconds. However, as the number of domains and allowable indices becomes large, say m*=50 and L=20, the time to test all possibilities is several years. Even using the simpler model of [SCHK75] and the theorems

presented there to limit the number of sets tested, the computation with m*=50 would take years to complete. If primary key and storage structure selection are added to the process, then the number of sets to be tested becomes $\sum_{i=1,L} 2i\binom{m^*}{i}$. The addition of attribute function indices to the selection process will increase the value of m* depending on the number of attribute functions being considered.

If, in addition to the previous extensions, we are interested in allowing concatenation of domains to be keys (as in combined indices of chapter 4), then the combinitorics become unmanageable for even very small values of m* (for m*=5 the number of possible simple and combined index sets is $2^{30}$). So in this case as well as the original problem with large m*, a heuristic is needed in order to find a solution in a reasonable amount of time.

We now present one possible heuristic for solving the selection problem.

Step 1 - Initialization

Given the set of potential keys, $D_p$ (which is all of the attributes plus any attribute functions being considered), partition $D_p$ into three mutually exclusive subsets, $D_a$, $D_b$, and $D_e$. $D_a$ is the set of all domains which are no longer under consideration and initially contains all attributes which have $p_i$=0. $D_b$ is the set of domains which have already been selected to be keys and is initially empty. $D_c$ is the set of candidate attributes (those attributes not yet

chosen but still under consideration) and is initially $D_p$-$D_a$.

Step 2 - Find next z best keys

Find the set D*, a subset of $D_c$ of size $\leq z$, such that D* union $D_b$ minimizes the cost function for all such sets.

Step 3 - Add the next z best keys to the current set.

If the addition of D* does not improve the cost of $D_b$ then terminate the algorithm, else set $D_b$ equal to the union of $D_b$ and D* and repeat steps 2 and 3.

The algorithm reduces the complexity of the problem by making the selection process an incremental one. There are several important points to note about an incremental algorithm such as this:

1. Obviously the choice of the parameter z is very important. If z is chosen to be large then the performance of the algorithm will approach that of complete enumeration as will its complexity. If z is chosen to be 1 then the algorithm will add only one new index to the set on each iteration. It is very easy to construct cases where adding one index at a time will produce results which are far from optimal (the example given in section 5.3 is one such case). With z chosen to be 3 or 4, cases can still be found where the algorithm will make bad choices, however, such cases require a complicated type of dependency to exist between several attributes.
2. The algorithm can be used to extend the index set after a period of time has elapsed. That is, a set of indices can

be chosen and then statistics may be gathered over some additional period of time. When the second set of statistics are examined, the decision process can consider the question of adding or removing attributes from the set. In either case, step 1 is skipped and replaced by initializing $D_b$ to be the current set of indices. Step 2 as given above then allows the addition of z more keys to the set. Removal of the z least important keys from the current set is accomplished by finding the best D* which is a subset of $D_b$ and of size $|D^*|-z$.

3. Since indices are added incrementally, an optimization over all relations in the data base can be achieved as opposed to the optimization previously described which works only over a single relation. To get some inter-relation optimization, step 2 is performed once for each relation. Then the improvement in cost of D* union $D_b$ over $D_b$ alone is noted for each relation. The relation having the greatest improvement in cost is the one for which step 3 is performed and those indices are actually created. Step 2 is then performed again for the relation selected and the process of selecting the most improved relation continues until a system wide limit on the number of secondary indices is reached. More sophisticated choices can be made by considering both increase in performance and the size of secondary indices.
4. In cases where $D_c$ is very large, a modification to the algorithm can further reduce the number of sets tested. Be-

fore step 2 is performed, each attribute in the set $D_c$ can be given a rating which indicates the likelihood that it will be chosen in the set of the next z best keys. This likelihood rating would be a function of one or several of the parameters of the cost function. The set $D_c$ would then be divided into two parts; the y most likely which would be used in step 2 and the remainder which would be saved for later consideration. Now, by adjusting the values of the parameters y and z, the number of cases tested in step 2 can be kept as small as necessary.

The algorithm suggested here is one of several possible methods of providing an approximate, yet fast solution to the selection problem. We suggest that more work is required in designing and testing algorithms which provide an even more accurate model of the data base environment.

## 5.7 Performance Monitoring

Any method of key selection which is based on query statistics makes the implicit assumption that past queries are representative of future ones. In many cases, this is a good assumption, however, it is always somewhat inaccurate. Along with this assumption, each approach to index selection makes other assumptions which make solution of the problem more tractable. Each assumption increases the chances that the optimal solution to the

stated problem will be non-optimal for the real problem of selecting the set of indices which best satisfy future queries. It is for this reason that some performance measure must be provided which can determine if the indices selected are, in fact, being utilized as expected. The important point to note is that no matter what optimization problem is solved and whether the solution is exact or approximate, the problem statement will have built into it certain assumptions which will make any solution only an approximate solution. It is because of the approximate nature of all solutions that performance monitoring is required.

A relation can easily be maintained which keeps up to date an indication of the utility of all indices in the data base. Such a relation might simply have the form INDEX_USAGE (INDEX_NAME, VALUE). A tuple is maintained for each index and whenever the index is used in query processing the number of accesses saved are added to the VALUE attribute. When the index is updated the number of accesses necessary to perform the update are subtracted from VALUE. The VALUE reflects the actual performance of the index set. This VALUE will give a good indication of which indices are useful in repetitious processing steps such as join terms where an index is used repeatedly during tuple substitution. The INDEX_USAGE relation allows for an easy comparison of the worth of indices on different relations. Also, this provides a good way to compare indices that were created by a dynamic decision to those created by the index selection procedure.

# CHAPTER 6

## Conclusions and Future Research

This dissertation has examined the choice of storage structures for relational data base management systems. We have described a set of structures to store data relations, a set of auxiliary relations, and a strategy for storage structure selection. In this chapter, we briefly summarize the highlights of the work and indicate directions for future research.

### 6.1 Storage Structures for Data Relations

In chapter 2, we specified three desirable conditions to be met by key to address functions. Since no function could be found which satisfied all three of the conditions independent of the data stored, a set of structures which would satisfy some of the conditions was examined. Randomizing functions were chosen as a structure to be used when order preservation was not important. For cases where order preservation was required, a generalized directory structure was introduced which performs at least as well as (and often better than) either normal directory structures or simple order preserving functions. Generalized directories provide a continuum of functions between simple order preserving functions at one extreme and normal directories at the other. An area which may prove fruitful for future research is

the set of functions which provide a continuum between simple order preserving functions and randomizing functions. It seeems likely that a class of functions exists which are more order preserving than randomizing functions yet provide a better distribution of keys across address space than do simple order preserving functions.

We described an algorithm for creating generalized directories which requires only a single pass over the data relation. Several experiments using the algorithm on a variety of data were described. These experiments indicate that for nearly uniform data, a generalized directory can be produced which requires only a small fraction of the number of parameters in a normal directory. For non-uniform data, a large reduction in directory size is possible if additional space is available and if some amount of page overflowing is allowed.

Next, the question of reorganization of directory structures was considered. Under a wide variety of circumstances, static directory structures were found to be superior to dynamic (continually reorganizing) structures. The reorganization of generalized static directories seems to be another promising area for future work. Since generalized directories are defined in segments, it appears that a method of partial reorganization could be devised. In such a scheme, only those segments of the directory which were in need of reorganiztion would be affected (i.e. only part of the directory would be modified).

## 6.2 Auxiliary Structures

In chapter 4 we were interested in finding information which could be stored in auxiliary relations which would be useful in speeding access to data relations. The approach used was to start with the simplest element of the query, the attribute, and work toward successively more complex pieces of the query, at each step considering an auxiliary structure to speed processing of that step. Secondary indices on simple attributes and attribute functions were found to be useful and easily maintainable. Combined indices are useful as an alternative to list processing techniques on several simple indices. Multi-relation predicates are costly to update but may be useful in retrieve-only situations where the predicate greatly restricts the cross product space of the relations involved. A more detailed evaluation of the tradeoffs involved with multi-relation links is currently under way [STON75]. An interesting question to consider in this context is whether a network data base foundation can be used as a basis for implementing an efficient relational system.

Also in chapter 4, a method of reusing aggregation information was introduced which requires very little overhead.

## 6.3 Storage Structure Selection Strategy

The first thing which must be realized when attempting to automate the selection of storage structures is the great complexity

of the problem. The approach that we have taken is to start with a small part of the problem, secondary key selection, and build upon it. A more widely applicable model of queries than that previously used was presented which resulted in a cost function that provides improved secondary index selection. A method for obtaining the parameters of the model was described in terms of a set of statistical relations and the queries necessary to extract the desired parameters. Extensions to the selection process were described for attribute function indices, primary key selection and primary storage structure type selection (i.e. randomized or directory). The cost of the selection process was found to be reasonable when a small to moderate number of keys were under consideration. When there are a large number of keys to be considered, it appears that heuristics are required. One such heuristic was suggested which is an incremental selection process. However, much more work is needed in developing good algorithms which use a more accurate model of the data base and limit substatially the number of possible solutions tested.

Finally a method was described to monitor the performance of the key selection process and remove poorly performing secondary indices. This too is an area in which more work is required. There must be methods to measure the performance of each storage structure in order that poorly performing structures can be identified.

We have taken a first step in examining the alternatives for

storage structures to support a relational data base system. Many assumptions have been made in order to make this complex problem tractable. The need now is for extensions which provide a more accurate reflection of the data base environment.

# APPENDIX A

## QUEL

QUEL (QUEry Language) has points in common with Data Language/ALPHA [CODD71], SQUARE [BOYC73] and SEQUEL [CHAM74] in that it is a complete [CODD72] query language which frees the programmer from concern for how data structures are implemented and what algorithms are operating on stored data. As such it facilitates a considerable degree of data independence [STON74b]. The sample relation from chapter one is used in the following set of examples which illustrate the QUEL language. For a detailed description of the language see [HELD75a] and for information on using the language in INGRES see [STON75a, ZOOK75].

A QUEL interaction includes at least one RANGE statement of the form:

RANGE OF variable-list IS relation-name

The symbols declared in the range statement are variables which will be used as arguments for tuples. These are called TUPLE VARIABLES. The purpose of this statement is to specify the relation over which each variable ranges.

Moreover, an interaction includes one or more statements of the form:

Command Result-name( Target-list )

WHERE Qualification

Here, Command is either RETRIEVE, APPEND, REPLACE, or DELETE. For RETRIEVE and APPEND, Result-name is the name of the relation which qualifying tuples will be retrieved into or appended to. For REPLACE and DELETE, Result-name is the name of a tuple variable which, through the qualification, identifies tuples to be modified or deleted. The Target-list is a list of the form

Result-domain = Function, ...

Here, the Result-domain's are domain names in the result relation which are to be assigned the value of the corresponding function.

The following suggest valid QUEL interactions. A complete description of the language is presented in [HELD75a].

Example A.1 Find the age of employee Jones

```
RANGE OF E IS EMPLOYEE
RETRIEVE INTO W(AGE = 1975 - E.BIRTH)
WHERE E.NAME = 'Jones'
```

Here, E is a tuple variable which ranges over the EMPLOYEE relation and all tuples in that relation are found which satisfy the qualification E.NAME = 'Jones'. The result of the query is a new relation, W, which has a single attribute, AGE, that has been calculated for each qualifying tuple. If the result relation is omitted, qualifying tuples are printed on the user's terminal. Also, in the Target-list, the 'Result-domain =' may be omitted if Function is of the form Variable.Attribute (i.e. NAME = E.NAME

may be written as E.NAME - see example A.6).

Example A.2 Insert the tuple (Jackson, candy, 13000, Baker, 1945, 1975) into EMPLOYEE.

```
APPEND TO EMPLOYEE(NAME = 'Jackson', DEPT = 'candy',
                   SALARY = 13000, MGR = 'Baker',
                   BIRTH = 1945, START = 1975)
```

Here, the result relation EMPLOYEE is modified by adding the indicated tuple to the relation.

Example A.3 Delete the information about employee Jackson.

```
RANGE OF E IS EMPLOYEE
DELETE E WHERE E.NAME = 'Jackson'
```

Here, the tuples corresponding to all employees named Jackson are deleted from EMPLOYEE.

Example A.4 Give a 10 percent raise to Jones

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY BY 1.1 * E.SALARY)
WHERE E.NAME = 'Jones'
```

Here, E.SALARY is to be replaced by 1.1*E.SALARY for those tuples in EMPLOYEE where E.NAME = 'Jones'. (Note that the keywords IS and BY may be used interchangeably with '=' in any QUEL statement.)

Also, QUEL contains aggregation operators including COUNT, SUM, MAX, MIN, AVG and the set operator SET. Two examples of the use of aggregation follow.

Example A.5 Replace the salary of all toy department employees by the average toy department salary.

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY BY AVG(E.SALARY WHERE E.DEPT = 'toy') )
        WHERE E.DEPT = 'toy'
```

Here, AVG is to be taken of the salary attribute for those tuples satisfying the qualification E.DEPT = 'toy'. Note that AVG(E.SALARY WHERE E.DEPT= 'toy') is scalar valued and consequently will be called an AGGREGATE. More general aggregations are possible as suggested by the following example.

Example A.6 Find those departments whose average salary exceeds the company wide average salary, both averages to be taken only for those employees whose salary exceeds $10000.

```
RANGE OF E IS EMPLOYEE
RETRIEVE INTO HIGHPAY(E.DEPT)
WHERE   AVG(E.SALARY BY E.DEPT WHERE E.SALARY > 10000)
        >
        AVG(E.SALARY WHERE E.SALARY > 10000)
```

Here, AVG(E.SALARY BY E.DEPT WHERE E.SALARY>10000) is an AGGREGATE FUNCTION and takes a value for each value of E.DEPT.

This value is the aggregate AVG(E.SALARY WHERE E.SALARY>10000 AND E.DEPT = value). The qualification expression for the statement is then true for departments for which this aggregate function exceeds the aggregate AVG(E.SALARY WHERE E.SALARY>10000).

# APPENDIX B

## Decomposition

The basic mechanism of processing statements in QUEL now follows. All update statements are processed into one or more RETRIEVE statements followed by a sequence of calls to the access methods (see Appendix C) to insert, delete or modify tuples. A RETRIEVE statement with more than one tuple variable is decomposed into a sequence of RETRIEVE statements each with a single tuple variable as described in [HELD75a]. The mechanism used is one of "tuple substitution". Here, we describe the algorithm for aggregate free interactions

Consider a query involving one or more tuple variables $X = (X_1,\ldots,X_N)$ with a range $R_1x\ldots xR_N$. Denote the qualification by $Q(X)$ and suppose $Q(X)$ is expanded into conjunctive normal form so that it consists of clauses connected by AND with each clause containing atomic formulas connected by OR. An atomic formula can contain only the boolean operator NOT.

Algorithm

1. stop if query has only a single variable

2. For each variable, say $X_1$ with Range $R_1$, collect all attributes which depend on $X_1$ and all clauses in the qualification which depend only on $X_1$. Say $D_1,\ldots,D_k$ are the

attributes and the clauses put together yield $Q1(X_1)$.

Issue the query:

RANGE OF $X_1$ IS $R_1$

RETRIEVE INTO $R_1'(X_1.D1,\ldots,X_1.Dk)$

WHERE $Q1(X_1)$

3. Replace the range $R_1$ in the original query by $R_1'$. The purpose of 2. and 3. is to limit each tuple variable to as small a relation as possible before continuing to step 4.

4. Take the variable with the fewest tuples in its range and substitute in turn the values of its tuples. This reduces the number of variables by one. After each substitution repeat 1.-3.

Step 4 is called tuple substitution and represents the most time-consuming step for multivariable queries. The choice of which variable to substitute for is critical. Our criterion (the one with the fewest tuple variables) is by no means optimal in general.

In this manner a multivariable query is reduced to a sequence of one-variable queries and calls to the access methods to obtain tuples for substitution. A one variable query is interpreted by a "one-variable query processor" (OVQP). This processor accesses tuples from the indicated relation one at a time, checks if the qualification is true for that tuple and if so assembles the target list attributes and inserts them into the result relation.

Besides interpreting the qualification and target list, this processor must:

1. ascertain if any secondary indices [STON74c] can profitably be used to speed access.

2. attempt to restrict the number of tuples accessed to less than all tuples in the relation.

The above description of the decomposition process works for any aggregate free QUEL query. The actual decomposition process used in INGRES uses several other techniques to break apart queries in such a way as to reduce the processing complexity.

# APPENDIX C

## Access Methods

To find all the tuples in a relation which satisfy the indicated qualification, the one variable query processor (OVQP) must either access and test all tuples in the relation or else must determine that only a subset of the relation need be tested with knowledge that the remainder of the relation can not satisfy the qualification. One way the OVQP might make such a determination is indicated in the following example. If the EMPLOYEE relation is sorted on increasing values of the SALARY domain, then in processing the query:

```
RANGE OF E IS EMPLOYEE
RETRIEVE E.NAME
WHERE E.SALARY < 10000 AND E.MGR = 'Jones'
```

the OVQP can stop testing tuples as soon as a tuple is encountered which has the SALARY domain greater than 10000. Depending on the particular storage structure which is in use for a given relation and the domains specified in the qualification, the OVQP may or may not be able to limit the number of tuples examined. Instead of having the OVQP and higher level software be concerned with this problem, a relational access method interface language (AMI) has been implemented. This language frees higher level software from details of actual storage structures and thus allows restructuring of relations for more efficient operation as

interaction conditions change. It has points in common with Gamma Zero [BJOR73], XRM [LORI74], and ZETA [CZAR75]. Relation access and update through AMI is accomplshed in the following manner.

1. A scan of a relation is begun by using the FIND statement to supply any information in the qualification which might be of help in limiting the range of the scan. FIND examines the information provided, and in conjunction with a knowledge of the storage structure used to implement the relation, determines starting and ending points for the scan.

2. Begining from the starting point tuples are accessed, one at a time, using the GET statement until the ending point is reached where GET returns an end of scan condition. The programmer may not assume that the tuples will be returned in any particular order.

3. Each tuple has a unique identifier called the tuple id (TID) which is returned with the tuple. This tuple id may be used to refer back to the tuple for re-access or updating (usually done after the qualification has been tested for the tuple).

4. INSERT, DELETE, and REPLACE statements are supported for all storage structures and respectively, add one new tuple to a relation, remove one tuple, or change the value of an exist-

ing tuple. When using REPLACE or DELETE the user must supply a TID to indicate which tuple is to be affected.

5. Apart from scan retrieval, GET also supports direct retrieval of tuples given a TID. This function is used in supporting secondary indices. Briefly, a secondary index is useful for limiting the number of tuples accessed in cases where a value for the primary domain (i.e. the domain used for ordering, for example SALARY above) is not present in the qualification. A secondary index is a relation which has one or more domains from the original relation along with a pointer domain which is an identifier of a tuple in the indexed relation. For instance, if SALARY is the ordering domain in EMPLOYEE it might be useful to have a secondary index on NAME. To build the secondary index all that is needed is a query of the form:

```
RANGE OF E IS EMPLOYEE
RETRIEVE INTO NAMEINDEX(E.NAME,PTR=E.TID)
```

This relation may then be stored in a structure which has NAME as the primary (ordering) domain. When a query on the EMPLOYEE relation specifies a value for NAME, the OVQP may access tuples in the NAMEINDEX relation and use the domain PTR as a TID to be supplied to GET which will return the corresponding tuple in the EMPLOYEE relation. Although two relations must be used to access tuples, a costly scan of the whole EMPLOYEE relation may be avoidable.

For AMI to support a new storage structure the following must be done.

1. A correspondence must be defined between a TID and a physical position in the structure.

2. There must be a linear ordering defined on TID's so that successive calls to GET will return all tuples in the relation.

3. FIND, GET, REPLACE, DELETE, and INSERT functions must be implemented for the new structure.

## References

ASTR74 Astrahan, M.M. & Chamberlin, D.D., "Implementation of a Structured English Query Language", IBM Research Report RJ-1464, Oct. 1974.

BACH74 Bachman, C., "The Data Set View vs. The Relational View", Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

BAYE70 Bayer, R. & McCreight, E., "Organization and Maintenance of Large Ordered Indices", Proc. 1970 ACM-SIGFIDET Workshop on Data Description, Access and Control, Houston, Texas, Nov. 1970.

BJOR73 Bjorner, D. & Codd, E.F. & Deckert, I.L. & Traiger, I.L., "The Gamma Zero n-ary Relational Data Base Interface: Specifications of Objects and Operations", IBM San Jose Research Report RJ1200, Apr. 1973.

BLIE67 Blier, R.E., "Treating Hierarchical Data Structures in the SDC Time-Shared Data Management System (TDMS)", Proc. ACM 22nd Natl. Conf., pp. 41-59, 1967.

BLIE68 Blier, R. & Vorkaus, A., "File Organization in the SDC Time Shared Data Management System (TDMS)", Proc. 1968 IFIP Congress, 1968.

BOYC73 Boyce, R. & et. al., "Specifying Queries as Relational

Expressions: SQUARE", IBM Research, San Jose, Ca., RJ 1291, Oct. 1973.

CHAM74 Chamberlin, D. & Boyce, R., "SEQUEL: A Structured English Query Language", Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

CHAM75 Chamberlin, D.D. & Gray, J.N., & Traiger, I.L., "Views, Authorization and Locking in a Relational Data Base System", Proc. 1975 NCC, pp. 425-430, AFIPS Press, May 1975.

CODA71a Committee on Data Systems Languages, "CODASYL Data Base Task Group Report", ACM, New York, 1971.

CODD70 Codd, E.F., "A Relational Model of Data for Large Shared Data Banks", CACM, Vol. 13 No. 6, pp. 377-387, June, 1970.

CODD71 Codd, E.F., "A Data Base Sublanguage Founded on the Relational Calculus", Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, CA, Nov. 1971.

CODD71b Codd, E.F., "Normalized Data Base Structures: A Brief Tutorial", Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, CA, Nov. 1971.

CODD72 Codd, E.F., "Relational Completeness of Data Base Sublanguages", Courant Computer Science Symposium 6, May 1972.

CODD72a Codd, E.F., "Further Normalization of the Data Base Relational Model", Courant Computer Science Symposium 6, Prentice-Hall May 1971.

CODD74 Codd, E.F. & Date, C.J., "Interactive Support for Non-Programmers, The Relational and Network Approaches", Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

CODD74a Codd, E.F., "Seven Steps to Rendevous with the Casual User", Proc. IFIP TC-2 Working Conference on Data Base Management Systems, Cargese, Corsica, Apr. 1974.

CZAR75 Czarnik, B. & Schuster, T. & Tsichritzis, D., "ZETA: A Relational Datat Base Management System", Proc. ACM-PACIFIC 75 Conf., pp. 21-25, Apr. 1975.

DATE74 Date, C.J. & Codd, E.F., "The Relational and Network Approaches: Comparison of the Aplication Programming Interfaces", Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

DEUT75 Deutscher, R.F. & Tremblay, J.P., & Sorenson, P.G., "Distribution-Dependant Hashing Functions and their Characteristics", Proc. 1975 ACM-SIGMOD Workshop, May 1975.

FEHR75 Fehr, E.S., "A Cost Study of Directory Structures for Ordered Files", Master's Thesis, University of Texas at Austin, Jan. 1975.

GOTT75 Gottlieb, D. et. al., "A Classification of Compression

Methods and their Usefulness in a Large Data Processing Center", Proc. 1975 NCC, pp. 453-458, AFIPS Press, May 1975.

HELD75a Held, G.D. & Stonebraker, M. & Wong, E., "INGRES - A Relational Data Base Management System", Proc. 1975 NCC, AFIPS Press, 1975.

IBM66 IBM Corp., "OS ISAM Logic", IBM, White Plains, N.Y., GY28-6618.

IBM70 IBM Corp., "IMS/360 Applications Description Manual", IBM, White Plains, N.Y., GH-20-0765

KEEH74 Keehn, D.G. & Lacy, J.O., "VSAM Data Set Design Parameters", IBM Systems Journal, Vol. 13, No. 3, pp. 186-213, 1974.

KING74 King, W.F., "On the Selection of Indices for a File", IBM Research RJ-1341, San Jose, Jan. 1974.

KNUT73 Knuth, D., "The Art of Computer Programming, Vol. 3", Addison Wesley, Reading, Mass. 1973.

LORI74 Lorie, R.A., "XRM- An Extended (n-ary) Relational Memory", IBM Cambridge Scientific Center Tech. Rep. 320-2096, Jan. 1974.

LUCK74 Lucking, J., "Data Base Languages, in particular DDL Development at CODASYL", Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

LUM70 Lum, V.Y., "Multiattribute Retrieval with Combined Indices", CACM Vol. 13, No. 11, pp.660-665, Nov. 1970.

LUM71a Lum, V.Y. & Yuen, P.S.T. & Dodd, M., "Key-to-Address Transform Techniques: A Fundamental Performance Study on Large Existing Formatted Files", CACM Vol. 14, No. 4, pp. 228-239, Apr. 1971.

LUM73 Lum, V.Y., "General Performance Analysis of Key-to-Address Transformation Methods Using an Abstract File Concept", CACM Vol. 16, No. 10, pp. 603-612, Oct. 1973.

MAUR75 Maurer, W.D. & Lewis, T.G., "Hash Table Methods", ACM Computing Surveys, Vol. 7, No. 1, pp. 5-20, Mar. 1975.

MCDO74 McDonald, N. & Stonebraker, M. & Wong, E., "Preliminary Specification of INGRES", University of California, Electronics Research Laboratory, Memorandum No. M435-436, April 1974.

MCDO75 McDonald, N. & Stonebraker, M., "Cupid -- The Friendly Query Language", Proc. ACM-Pacific 75 Conf., Apr. 1975.

MORR68 Morris, R., "Scatter Storage Techniques", CACM Vol. 11, No. 1, pp. 38-44, Jan 1968.

MULL71 Mullin, J., "Retrieval-Update Speed Tradeoffs Using Combined Indices", CACM Vol.14, No.12, pp. 775-776, Dec. 1971.

PECH75 Pecherer, R.M., "Efficient Evaluation of Expressions in a Relational Algebra", Proc. ACM-Pacific 75 Conference, pp. 44-49, Apr. 1975.

RITC73 Ritchie, D. & Thompson, K., "The UNIX Time Sharing System", CACM Vol. 17, No. 7, pp. 365-375, July 1974.

RIVE74 Rivest, R.L., "Analysis of Associative Retrieval Algor-

ithms", IRIA Report No. 54, Feb. 1974.

ROTH74a Rothnie, J.B. & Lozano, T., "Attribute Based File Organization in a Paged Memory Environment", CACM Vol. 17, No. 2, Feb. 1974.

SCHK74 Schkolnick, M., "The Optimal Selection of Indices for Files", Research Report, Department of Computer Science, Carnegie-Mellon Univ., Nov. 1974.

SCHK75 Schkolnick, M., "The Optimal Selection of Secondary Indices for Files", Proc. 1975 SIGMOD Workshop, May 1975.

SIBL74 Sibley, E., "On the Equivalence of Data Base Systems", Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

SMIT75 Smith, J.M. & Chang, P.Y.T., "Optimizing the Performance of a Relational Data Base Interface", Proc. 1975 SIGMOD Workshop, May 1975.

STON72 Stonebraker, M., "Retrieval Efficiency Using Combined Indices", Proc. 1972 ACM-SIGFIDET Workshop on Data Description, Access and Control,

STON74 Stonebraker, M. & Wong, E., "Access Control in a Relational Data Base Management System by Query Modification", Proc. 1974 ACM National Conference, San Diego, Ca., Nov. 1974

STON74b Stonebraker, M, "A Functional View of Data Independence", Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May

1974.

STON74c Stonebraker, M., "The Choice of Partial Inversions and Combined Indices", International Journal of Computer and Information Sciences, Vol.3, No.2, 1974.

STON74d Stonebraker, M., "High Level Integrity Assurance in Relational Data Base Management Systems", Univ. of California, Berkeley, ERL Mem. No. M473, Aug. 1974.

STON75 Stonebraker, M. & Held, G.D., "Networks, Hierarchies, and Relations in Data Base Management Systems", Proc. ACM-Pacific 75 Conf., Apr. 1975.

STON75a Stonebraker, M.R., "Getting Started in INGRES - A Tutorial", University of California, Berkeley, ERL Mem. No. ERL-M518, Apr. 1975.

STON75b Stonebraker, M.R., "Using a Network Foundation for a Relational Data Base System", To appear.

TSIC75 Tsichritzis, D., "A Network Framework for Relational Implementation", University of Toronto, Computer Systems Research Group Report CSRG-51, Feb. 1975

WHIT74 Whitney, V.K.M., "Relational Data Management Implementation Techniques", Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

WHIT75 Whitt, J.D. & Sullenberger, A.G., "The Algorithm Sequential Access Method: An Alternative to Index Sequential", CACM Vol. 18, No. 3, pp. 174-176, Mar. 1975.

WONG75 Wong, E., "Decomposition - Query Processing in INGRES",

Private Communication, May 1975.

ZOOK75 Zook, W. & Youssefi, K. & Kreps, P. & Held, G. & Ford, J., "INGRES - Reference Manual", University of California, Berkeley, ERL Mem. No. ERL-M519, Apr. 1975.

## Related Bibliography

ASH68 Ash, W. & Sibley, E.H., "TRAMP, An Interpretive Associative Processor with Deductive Capabilities", Proc. ACM 23rd Natl. Conf., Brandon Systems Press, Princeton N.J., pp.143-156, 1968.

ASTR75 Astrahan, M.M. & Lorie, R.A., "SEQUEL - XRM, A Relational System", Proc. ACM-PACIFIC 75 Conf., pp. 34-38, Apr. 1975.

BACH65 Bachman, C.W., "Software for Random Access Processing", Datamation, pp. 36-41, Apr. 1965.

BACH73 Bachman, C.W., "The Programmer as Navigator", CACM Vol. 16, No. 11, pp.653-658, Nov. 1973.

BAER68 Baer, R.M. & Brock, P., "Natural Sorting over Permutation Spaces", Math. Comp. Vol. 22, pp. 385-410, 1968.

BELL66 Bell, C.J., "A Relational Model for Information Retrieval and the Processing of Linguistic Data", IBM Research Report RC1705, Yorktown Heights, New York, Nov. 1966.

BOYC73a Boyce, R. & Chamberlin, D., "Using a Structured English Query Language as a Data Definition Facility", IBM Research Laboratory, San Jose, Ca. RJ 1318, Dec. 1973.

BRAC72 Bracchi, G., "A Language for a Relational Data Base Management System", Proc. 6th Annual Princeton Conf. on

Info. Science and Systems, Mar. 1972.

BROW71 Browne, P. & Steinauer, D., "A Model for Access Control", Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, CA, Nov. 1971.

BUCH63 Bucholz & Werner, "File Organization and Addressing", IBM Systems Journal, Vol. 2, pp. 86-111, June 1973.

CHIA72 Chiang, T.C., "A File Structure for Large Data Bases", Ph.D. Dissertation, Univ. of California, Berkeley, 1972.

CHIL68 Childs, D.L., "Description of a Set Theoretic Data Structure", Proc. 1968 FJCC Part 1, pp. 557-564

CODA71 Committee on Data Systems Languages, "Feature Analysis of Generalized Data Base Management Systems", ACM, New York May 1971.

CODA74 Committee on Data Systems Languages, "Data Description Language", Handbook #112, U.S. Department of Commerce, January, 1974.

CODD74b Codd, E.F., "Recent Investigations in Relational Data Base Systems", Information Processing 74, North Holland, Amsterdam

COLL72 Collmeyer, A., "Implications of Data Independence on the Architecture of Data Base Management Systems", Proc. 1972 ACM-SIGFIDET Workshop on Data Description, Access and Control,

COPE74 Copeland, G.P. & Su, S.Y.W., "A High Level Data Sublanguage for a Context-addressed Segment-sequential Memory", Proc. 1974 ACM-SIGFIDET Workshop on Data

Description, Access and Control, Ann Arbor, Mich., May 1974.

DATE71 Date, C.J. & Hopewell, P., "File Definition and Logical Data Independence", Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, CA, Nov. 1971.

DATE71a Date, C.J. & Hopewell, P., "Storage Structure and Physical Data Independence", Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, CA, Nov. 1971.

DATE72 Date, C.J., "Relational Data Base Systems: A Tutorial", Proc. of the COINS-72 Symposium, Dec. 1972.

DELO73 Delobel, C. & Casey, R.G., "Decomposition of a Data Base and the Theory of Boolean Switching Functions", IBM Journal of Research and Development, Sep. 1973.

ESWA73 Eswaren, K., "Consecutive Retrieval Information System", Ph.D. Dissertation, Univ. of California, Berkeley, 1973.

EVER74 Everest, G., "Concurrent Update Control and Data Base Integrity", 1974 IFIP Conference on Data Base Management Systems, Cargese Corsica, April 1974.

FARL75 Farley, J.H.G. & Schuster, S.A., "Query Execution and Index Selection for Relational Data Bases", University of Toronto Technical Report CSG-53, March 1975.

FELD69 Feldman, J.A. & Rovner, P.D., "An Algol-Based Associative Language", CACM Vol.12, No.8, pp.439-449, Aug. 1969.

FREI70 Friedman, T., "The Authorization Problem in Shared Files", IBM Systems Journal Vol. 9 No. 4, 1970.

GOLD70 Goldstein, R.C. & Strnad, A.L., "The MacAims Data Management System", Proc. 1970 ACM-SIGFIDET Workshop on Data Description, Access and Control, Houston, Texas, Nov. 1970.

GRIE71 Gries, D., "Compiler Construction for Digital Computers", Wiley & Sons, New York 1971, pp. 315-317

GUST71 Gustafson, R.A., "Elements of the Randomized Combinatorial File Structure", Proc. Symp. on Information Storage & Retrieval, ACM, pp. 163-174, Apr. 1971.

HEAT71 Heath, I.J., "Unacceptable File Operations in a Relational Data Base", Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, CA, Nov. 1971.

HELD75 Held, G.D. & Stonebraker, M., "Storage Structures and Access Methods in the Relational Data Base Management System, INGRES", Proc. ACM-Pacific 75 Conf., Apr. 1975.

HOUS75 Housel, B.C. et. al., "DEFINE: A Non-Procedural Data Description Language for Defining Information Easily", Proc. ACM-PACIFIC 75 Conf., pp. 62-70, Apr. 1975.

HSIA70 Hsiao D. & Harray, F., "A Formal System for Information Retrieval from Files", CACM, Vol. 13, No. 2, pp. 67-73, Feb. 1970.

ISAA56 Isaac, E.J. & Singleton, R.C., "Sorting by Address Calculation", JACM Vol.3, No. 2, pp. 169-174, Feb. 1956.

JOHN74 Johnson, S.C., "YACC, Yet Another Compiler-Compiler", UNIX Programmer's Manual, Bell Telephone Labs, Murray Hill, N.J., July 1974.

JORD75 Jordan, D.E., "Implementing Production Systems with Relational Data Bases", Proc. ACM-PACIFIC 75 Conf., pp.39-43, Apr. 1975.

KERN74 Kernighan, B.W., "A Tutorial Introduction to the ED Text Editor", UNIX Programmer's Manual, Bell Telephone Labs, Murray Hill, N.J. July 1974

KNOT71 Knott, G.D. , "Expandable Open Addressing Hash Table Storage and Retrieval", Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, CA, Nov. 1971.

KNOT74 Knott, G.D., "Hashing Functions", British Computer Journal

KNUT69 Knuth, D., "The Art of Computer Programming, Vol 1.", Addison Wesley, Reading, Mass. 1969.

KRON65 Kronmal, R.A. & Tarter, M.E., "Cumulative Polygon Address Calculation Sorting", Proc. 20th National ACM Conf., ACM, pp. 376-384, 1965.

LEFK69 Lefkovitz, D., "File Structures for On-Line Systems", Spartan Press, Washington, D.C., 1969.

LEVI65 Levien, R.E. & Maron, M.E., "Relational Data File: A Tool for Mechanized Inference Execution and Data Retrieval", Rand Corp. Mem. RM-4793-PR, Santa Monica, CA, Dec. 1965.

LEVI67 Levien, R.E. & Maron, M.E., "A Computer System for Inference Execution and Data Retrieval", CACM Vol. 10, No. 11, pp. 715-721, Nov. 1967.

LEVI69 Levien, R.E., "Relational Data File: Experience with a System for Propositional Data Storage and Inference Execution", Rand Corp. Mem. RM-5947-PR, Santa Monica, CA, Apr. 1969.

LORI70 Lorie, R.A. & Symonds, A.J., "A Schema for Describing a Relational Data Base", Proc. 1970 ACM-SIGFIDET Workshop on Data Description, Access and Control, Houston, Texas, Nov. 1970.

LOWE68 Lowe, T.C., "The Influence of Data Base Characteristics and Usage on Direct Access File Organization", JACM Vol.15, No.4, pp.535-548, Oct. 1968.

LUM71 Lum, V.Y. & Ling, H., "An Optimization Problem on the Selection of Secondary Keys", Proc. ACM Natl. Conf. 1971, pp.349-356.

MCGE69 McGee, W.C., "Generalized File Processing", Annual Review in Automatic Programming Vol.5,No.13., pp. 77-149,Pergamon Press, New York, 1969.

MCIN73 McIntosh, S. & Griffel, D., "Data Management for a Penny a Byte", Computer Decisions, May 1973.

NIEV74 Nievergelt, J., "Binary Search Trees and File Organization", ACM Computing Surveys Vol. 6, No. 3, pp. 195-207, Sept. 1974.

NOTL72 Notley, M.G., "The Peterlee IS/1 System", IBM UK Scien-

tific Centre Report UKSC-0018, Mar. 1972.

OLLE68 Olle, T.W., "A Non-Procedural Language for Retrieving Information from Data Bases", IFIP Congress, Edinburgh, North-Holland, Amsterdam, Aug. 1968.

OWEN71 Owens, R., "Evaluation of Access Authorization Characteristicts of Derived Data Sets", Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, CA, Nov. 1971.

OWEN71b Owens, R., "Primary Access Control in Large Scale Time-Shared Decision Systems", Project MAC Report TR-89, M.I.T., Cambridge, Mass., July 1971.

PALE70 Palermo, F., "A Quantitative Approach to the Selection of Secondary Indices", Formatted File Organization Techniques, IBM Research, San Jose, Mar. 1970.

PALE72 Palermo, E.P., "A Data Base Search Problem", Proc. 4th International Symposium on Computers and Information Science, Miami Beach, Dec. 1972.

RAMA71 Ramamoorthy, C.V. & Blevins, P.R., "Arranging Frequency Dependent Data on Sequential Memories", Proc. 1971 SJCC, AFIPS, Vol. 38, pp. 545-556, 1971.

RAMI71 , "RAMIS - Users Manual", Mathematica, Inc., Princeton, New Jersey.

RISS73 Rissanen, J. & Delobel, C., "Decomposition of Files, a Basis for Data Storage and Retrieval", IBM San Jose Research Report RJ-1220, May 1973.

RITC74 Ritchie, D.M., "C Reference Manual", UNIX Programmer's

Manual, Bell Telephone Labs, Murray Hill, N.J. July 1974.

RITC74a Ritchie, D. & Thompson, K., "UNIX Programmer's Manual", Bell Telephone Labs, Murray Hill, N.J. June 1975

ROTH72 Rothnie, J.B., "The Design of Generalized Data Management Systems", Ph.D. Dissertation, Dept. of Civil Engr., M.I.T., 1972.

ROTH74 Rothnie, J., "An Approach to Implementing a Relational Data Base Management System", Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

SEVE72 Severence, D.G., "Some Generalized Modeling Structures for Use in Design of File Organizations", Ph.D. Dissertation, Univ. of Michigan, 1972.

SEVE74 Severance, D.G., "Identifier Search Mechanisms: A Survey and Generalized Model", ACM Computing Surveys Vol. 6, No. 3, pp.175-194, Sept. 1974.

SIBL73 Sibley, E. & Taylor, R., "A Data Definition and Mapping Language", CACM Vol.16, No.12, Dec. 1973.

STEU74 Steuert, J. & Goldman, J., "The Relational Data Management System: A Perspective", Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

SYMO70 Symonds, A.J. & Lorie, R.A., "A Schema for Describing a Relational Data Base", Proc. 1970 ACM-SIGFIDET Workshop on Data Description, Access and Control, Houston, Texas,

Nov. 1970.

TAHA72 Tahani, V., "A General Conceptual Framework for Information Retrieval Systems", Ph.D. Dissertation, Univ. of California, Berkeley, 1972.

TART68 Tarter, M.E. & Kronmal, R. A., "Estimation of the Cumulative by Fourier Series Methods and Application to the Insertion Problem", Proc. 23rd National ACM Conference, ACM, pp. 491-497, 1968.

WAGN73 Wagner, R.E., "Indexing Design Considerations", IBM Systems Journal, Vol. 12, No. 4, pp. 351-367, 1973.

WEIS69 Weissman, C., "Security Controls in the ADEPT-50 Time Sharing System", Proc. 1969 Fall Joint Computer Conference, Nov. 1969.

WHIT72 Whitney, V.K.M., "RDMS: A Relational Data Management System", Report CS 80, General Motors Research, Warren, Mich., Dec. 1972.

WONG71 Wong, E. & Chiang, T., "Canonical Structure in Attribute Based File Organization", CACM Vol.14, No.9, Sep. 1971.

ZLOO75 Zloof, M.M., "Query by Example", Proc. 1975 NCC, pp. 431-438, AFIPS Press, May 1975.