A PARAMETERIZED MODEL FOR SELECTING THE OPTIMUM FILE ORGANIZATION IN MULTI-ATTRIBUTE RETRIEVAL SYSTEMS

by

IRVING JACOB SHACHAT

S.B., Massachusetts Institute of Technology
(1968)

Submitted in partial fulfillment

of the requirements for the degree of

Master of Science

at the

Massachusetts Institute of Technology

June, 1974

Signature of Author _____
Alfred P. Sloan School of Management, May 10, 1974

Certified by _____
Thesis Supervisor

Accepted by _____
Chairman, Departmental Committee on Graduate Students

ABSTRACT

A PARAMETERIZED MODEL FOR SELECTING

THE OPTIMUM FILE ORGANIZATION

IN MULTI-ATTRIBUTE RETRIEVAL SYSTEMS

by

IRVING JACOB SHACHAT

Submitted to the Alfred P. Sloan School of Management on May 10, 1974, in partial fulfillment of the requirement for the degree of Master of Science in Management.

This thesis develops a general parameterized model that facilitates the comparison of different file organization techniques for a given multiple key information retrieval system. The model is based on minimizing the expected processing time of the data base in performing on-line retrieval and updating operations. The decision rules are a function of the relevant characteristics of the data base, the on-line queries, the storage devices, and the file organization techniques, as well as the relative breakdown of the processing requests between retrievals and various types of updating operations.

To demonstrate the use of the model, detailed timing formulas are developed for the retrieval and updating operations for three different file organizations: the Multilist system, the Inverted Index system, and the Cellular Serial system. Several examples are presented illustrating the application of the model to choosing among these three systems in a wide variety of specific situations. In the majority of these examples the model shows that the Inverted Index system would prove to be substantially more efficient than the other two systems.

Given these results an entire chapter is devoted to a discussion of a number of specific implementation alternatives available under the Inverted Index system and the relevant trade-offs between them. Several radical new file organization schemes that claim significant improvements in retrieval efficiency by clustering together records that are frequently retrieved together are also discussed.

A comprehensive review of the literature on file organizations for multiple key retrieval is included as part of the thesis, along with a complete bibliography on much of the theoretical material concerning both single and multiple key retrieval techniques.

Thesis Supervisor: Stuart Madnick
Title: Assistant Professor of Management

ACKNOWLEDGEMENTS

I wish to thank Dr. Stuart Madnick for his guidance and encouragement throughout this effort and Dr. Michael Scott-Morton for serving as Second Reader, when an administrative snafu had left me readerless.

I also wish to express my sincere gratitude to Frances Micklay for her careful typing and re-typing of this paper throughout its preparation. Finally, special thanks is due my new bride, Emily, for her encouragement, patience, and love that helped make this paper possible, and that certainly made it all worthwhile.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

By far the vast majority of all data files and data bases in use at the present time are organized for retrieval based on a single "key". A key may be defined as a data item which partially characterizes one or more records in the file, and by which these records can be accessed. For example, a personnel data file organized[1] by the key "Employee-Last-Name" could be readily used to locate the record of "HIGGENBOTTOM, TIMOTHY H.". Similarly, another data file containing the same information as the first, but organized by the key "Employee-Home-City", could be used to access the records of all employees who live in "BELMONT". In both cases, retrieval of records according to the primary key (the key on which the file was organized) is reasonably efficient, while retrieval on any other key generally requires a full serial search of the file to locate all qualified records.

Among the many file organizations that facilitate single key retrieval are the indexed sequential; the random access (including hash coding); the dense key-ordered files that permit binary searching and/or supplementary directories; the various

[1]To understand what is meant by "organized", picture the file as being sorted in order according to the specified key, with the retrieval of desired records being accomplished speedily by a binary search or by an associated directory. However, there are other single-key techniques, such as hash coding, that do not involve the ordering of the records in an intuitive manner.

types of chained (linked list), or ring structures; and the many variations of the hierarchical and tree-structured file organizations. These single key techniques largely form the basis for the more complex multiple key file organizations which are the subject of this thesis. However, the single key techniques per se and the relevant trade-offs between them have been well documented elsewhere and, except for a short review of the relevant literature in Chapter III, they will not be discussed further here.

An area of greater research interest and of tremendous practical interest is that of designing efficient systems for retrieving records using any of several different keys. A case in point might be a personnel data file from which it was desired to retrieve records by employee name, or department, or occupation, or salary level. Such multiple key file organizations can perform a retrieval operation on any desired key, and generally on a logical combination of keys, without necessitating a full file search. The most frequently used logic functions for combining keys are the Boolean operators AND, OR, and NOT. For example, a company with 50,000 employees that is staffing its new plant in Germany may wish to query its multi-key data base as follows: Retrieve the records of all employees who are either chemists or mechanical engineers, who have been with the company at least three years, who do <u>not</u> have a Ph.D, and who are willing to relocate.

A number of sophisticated file organization techniques have been implemented (or proposed) in the last few years to satisfy

queries such as the above for generalized data bases. Each of these techniques has its own special characteristics in terms of the speed of retrieval operations, the speed of various types of on-line updating operations (if permitted), the cost of the initial loading of the data base, the extent and complexity of the programming involved, the disk and core storage requirements, and the cost and frequency of reorganizations needed to maintain efficiency. Not surprisingly, there are several reasons why the selection of the best technique to use in a given situation is not generally obvious:

(1) Many of the relevant file organization characteristics are non-qualitative or are expressed in difficult-to-compare dimensions. For instance, how should one decide between a system that provides very quick retrieval and one that is slower, but that is easier to program, quicker in updating, and that consumes less disk storage?

(2) Even those characteristics that are potentially quantifiable may be quite difficult to estimate in a given situation. For example, the average retrieval and updating speeds of most complex file organizations depend on so many factors that even experienced data base designers find it difficult to provide accurate pre-implementation predictions of retrieval times.

(3) Users' requirements change over time. A data base that was designed to be efficient in answering various single key queries (which the users said would predominate) may turn out to be used for complex, multi-key queries. A data base that was originally planned for 100,000 records and 5 keys per record may expand with

its initial success to 1,000,000 records and 25 keys per record. Often, the data base is not as efficient in handling the new, unplanned requirements as another technique that was originally passed over.

(4) Many of the proposed techniques have never been implemented in actual data bases. The basic reason for this is the prohibitive cost involved in doing research with "real-world" data bases. In fact, a number of the more promising new techniques have never been implemented even on artificially constructed data bases.

The principal objective of this paper is to develop a comprehensive, parameterized model that facilitates the direct comparison of various file organization techniques. The model is based on minimizing the average on-line response time of the data base in performing both retrieval and updating operations (including record insertions, deletions, and modifications). The model incorporates such factors as the length and complexity of the queries, the size and key dispersion of the data base, the characteristics of the direct access storage device, and the percentage breakdown of the on-line operations between retrievals and the various types of updating operations.

The obvious benefit of such a model is its ability to predict prior to implementation the expected performance of the different file organization techniques, and to allow the selection of the optimum storage structure for any given data base. Another possible use of the model would be as a critical module within a truly generalized information retrieval system.

Recognizing that no one file organization is best in all cases,[2] Professors S. Madnick of M.I.T. and A. Cardenas of U.C.L.A. (13) have independently proposed such a generalized information retrieval system that, among its many advanced features, would contain a library of all undominated file organization techniques. Using a model such as the one developed in this paper, the system would choose the most appropriate file structure and automatically organize the data base according to that structure. Moreover, the system would continually monitor the performance of the data base it had created; if the performance differed significantly from the predicted performance — either because of misestimated parameter values or because of an actual change in the data base characteristics — the system would restructure the data base according to a more efficient file organization (taking into account, of course, the cost of restructuring). As new file organizations and hybrids of existing techniques were developed and perfected, they could be added to the system library and incorporated into the decision-making model.

The development of this paper has been designed to parallel the way one would remove the successive layers of an onion: each chapter builds on the previous chapters, covering in more detail points or problem areas that were glossed over earlier. The principal aim of this approach is gradually to introduce the unsophisticated reader to increasingly more complex examinations of

[2]In fact, it is generally true that file organizations that perform the best under retrieval are often the worst under update (Lefkovitz (49)).

the multi-key problem; and in some ways the progression parallels the developing sophistication of the multi-key techniques.

Chapter II of the thesis introduces several file organization schemes and discusses how each handles various multi-key retrieval problems, commenting on their respective advantages and limitations. Chapter III begins with a brief, but fairly complete, summary of the relevant literature on single key techniques. It then attempts to provide an overview of most of the accepted theoretical, and a good deal of the practical, literature on multi-key retrieval, leaving for discussion in Chapter VII some of the more interesting but as yet untested techniques.

Chapter IV develops the basic retrieval timing formulas used in the model for the implementations discussed in Chapter II, and presents some basic hypotheses concerning retrieval efficiency. Chapter V develops the basic timing formulas for the various types of on-line updating operations, and synthesizes the results into a general parameterized model. The model is then applied to several concrete situations, allowing further generalizations concerning preferred file organizations.

Chapter VI covers a lot of the micro-level implementation problems that were bypassed earlier, including alternative methods for decoding individual keys within the directory, techniques for maintaining hierarchical associations within the data base records, and methods for performing list intersections in core. Chapter VII examines some of the proposed ideas for

improving efficiency by clustering together records that are commonly retrieved together. Techniques that attempt to promote clustering have been developed as extensions to existing techniques and also as radically new techniques. Chapter VIII summarizes the results and provides suggestions for future research.

CHAPTER II

FILE ORGANIZATION TECHNIQUES

This section discusses several of the better documented multiple key file organization techniques that have been reported on by practitioners in the field. Two of the techniques — the Multilist and the Inverted Index file organizations — have achieved widespread usage. The other three techniques are off-shoots of the first two, and although they have less "real world" experience, they appear to have been fairly well analyzed in the literature. In all cases it is assumed that the data is stored on disk (or other direct access mass storage device) and is transferred to core memory for processing.

Much of the discussion of specific retrieval techniques in this chapter is derived from papers by Lefkovitz (49), Martin (57), and Siler (83). In particular, five of the six figures in this chapter (all except Figure 3) are taken directly from Lefkovitz's book. Although this chapter goes below the macroscopic level in discussing retrieval methodology and specific advantages and limitations, it attempts to stay away from very detailed considerations that might obscure the main points. Chapter VI reconsiders some of these same file organizations to delineate some specific problem areas and to discuss some of the alternative solutions that have been proposed.

2.1 The Multilist File Organization

The Multilist File organization, which has also been called

the Multi-linked List organization and the Multiple Threaded List organization, has been popping up in the literature for almost a dozen years. An outgrowth of the standard linked-list or chained data structures, the Multilist data base has two basic components: a key directory and a list-structured file. Each unique key by which the file is to be accessed has a three-part entry in the key directory: the key itself, a pointer to the first record on the linked list for that key, and a number telling how many records are on that list. Within the first record on the list is a pointer to the second record, and so on. Thus, a data base record having twenty keys would also have twenty pointers to the next record on each of the respective key lists.

Multilist performs a typical key search by first accessing all the relevant keys from the key directory. If an "AND" operation is to be performed it chooses the shortest of the key lists and moves from record to record on that list identifying which records on that list also contain the other required keys. If an "OR" operation is to be performed, each list associated with a key in the query product must be processed.[3] Query conjunction involving the "NOT" operation (e.g., A AND B AND NOT C) are handled by following the shortest list of the nonnegated

[3]The Boolean queries are assumed to have been translated into disjunctive normal form, i.e., a sequence of query products or conjunctions connected by OR's where each product is composed of operands connected by AND's.

keys. A query conjunction that does not contain any nonnegated keys is impractical because it requires that an excessive number of lists be processed.

An example might be a query involving keys W AND X ($W \wedge X$) in Figure 1. Since the list for key X is shorter than the list for key W, the search progresses down the list for key X. The first record accessed is stored at address A3. The address of the next record to be accessed is contained in record A3. The search progresses to the end of the list for key X. Two records are found on the list for key X which are also on the list for key W. These two records are retrieved as responses to the query.

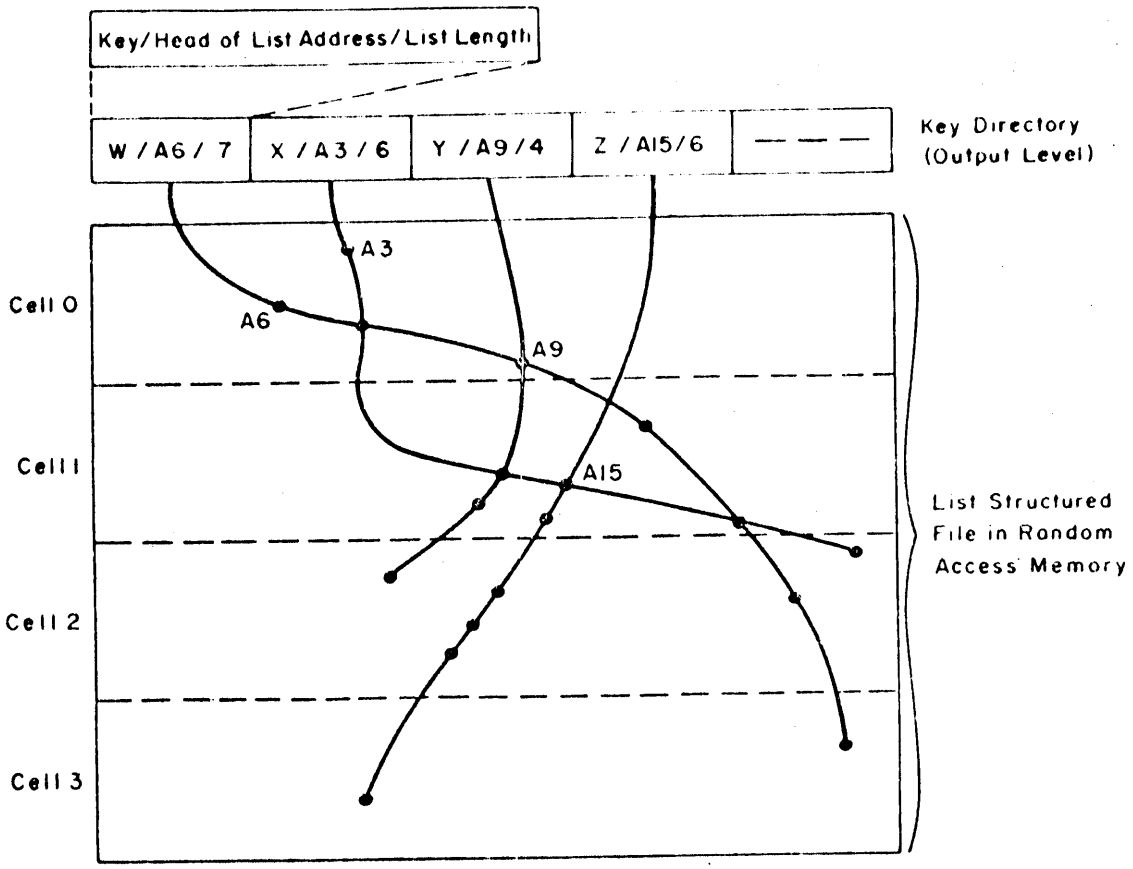


Figure 1: The Multilist File Organization

The greatest disadvantage of the Multilist organization is that in order to respond to a conjunction of terms (such as $W \wedge X$), it must access and transfer to core all records on the shortest list, even though it is only the intersection of these lists (which may be much smaller) that satisfies the query. In the case of $W \wedge X$, the ratio of the number of records satisfying the query to the number of records retrieved was 1 to 6. In practice, if list lengths are several hundred (or thousand) long and conjunctions contain many keys, this ratio may be on the order of a few hundredths (or thousandths), which is highly inefficient.

A reflection of this inefficiency is also found in the low quality presearch retrieval statistics. The shortest list length in a query conjunction, or the sum of such list lengths for a logical sum of products, is the closest upper bound on the number of retrievals that a Multilist organization can provide prior to the file search. The other structures to be described yield considerably better presearch statistics.

The principal advantages of the Multilist file organization are programming simplicity and updating speed and flexibility. On-line record deletions are handled by setting a Record Delete Bit to 1. The actual purging of deleted records can take place during periodic batch-mode file regenerations. Record addition is also easily accomplished by putting the record at the head of each of its key lists, thus leaving the remainder of the key lists undisturbed.

2.2 The Inverted Index File Organization

In the Inverted Index (or Inverted List) file organization, each unique key is associated with a list of the addresses of the records which are indexed under that key. For complicated queries this organization performs the logical operations directly on the inverted lists, and thus it needs to retrieve only those records that satisfy all conditions specified in the query.

For example, a query for W $\wedge$ X in Figure 2 begins by accessing the inverted lists of addresses for key W and for key X. These lists are intersected and it is found that the only addresses in common are A7 and A19. These two records are then retrieved as responses to the query. Since only those records that satisfy the query requirements are retrieved, the total number of record retrievals is usually significantly lower than for the Multilist organization. Note that even before it accessed the desired records, the inverted index can inform the user of the exact number of records that satisfy his query. A more complete view of the handling of files of the Inverted Index system and the way it handles queries is shown in Figure 3.

In the Inverted Index file organization, the logical conjunction of nonnegated terms (e.g., A $\wedge$ B $\wedge$ C) is accomplished by list intersection in core; the disjunction of nonnegated terms (e.g., A $\cup$ B $\cup$ C) is accomplished by list merging, and the conjunction of a nonnegated key with a negated key is accomplished by removing from the nonnegated key's list of addresses, all those that appear on the negated key's list. The retrieval of

an isolated negated key term list is usually tantamount to a serial search of the entire file and, hence, would be performed as such if ever required, since key list lengths are normally less than a few percent of the entire file size. Most real-time list structured systems would disallow such a query, although it could be relegated to the batched mode of retrieval.

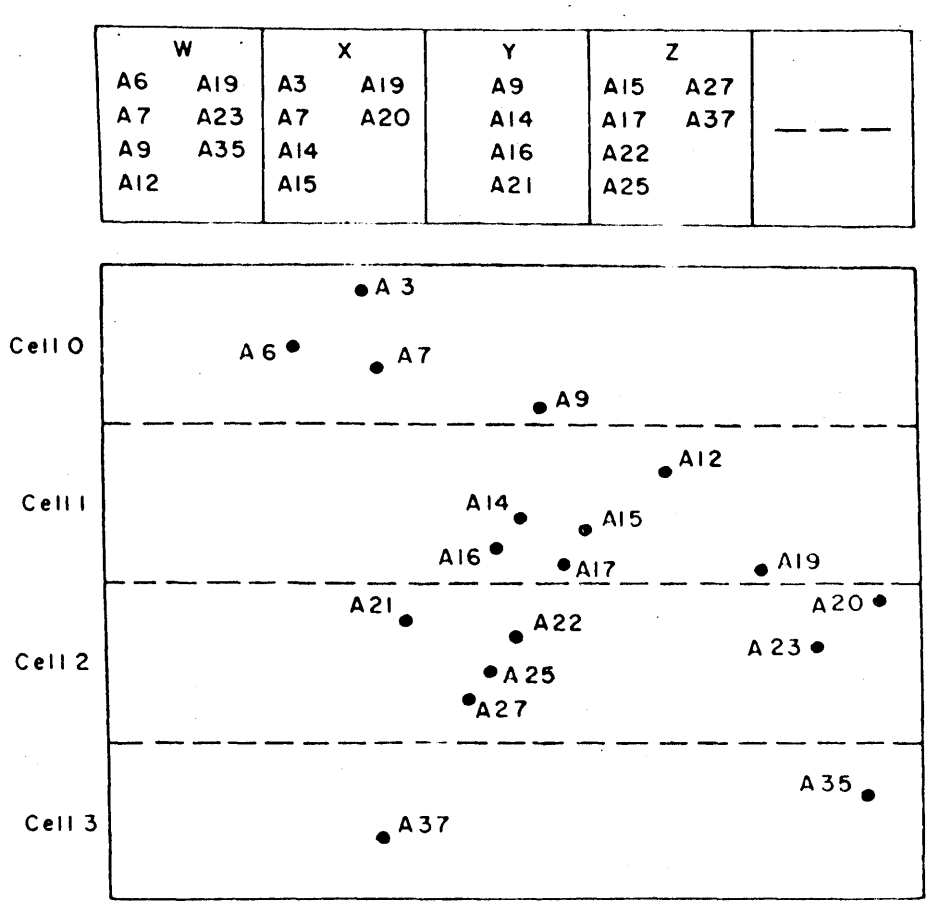
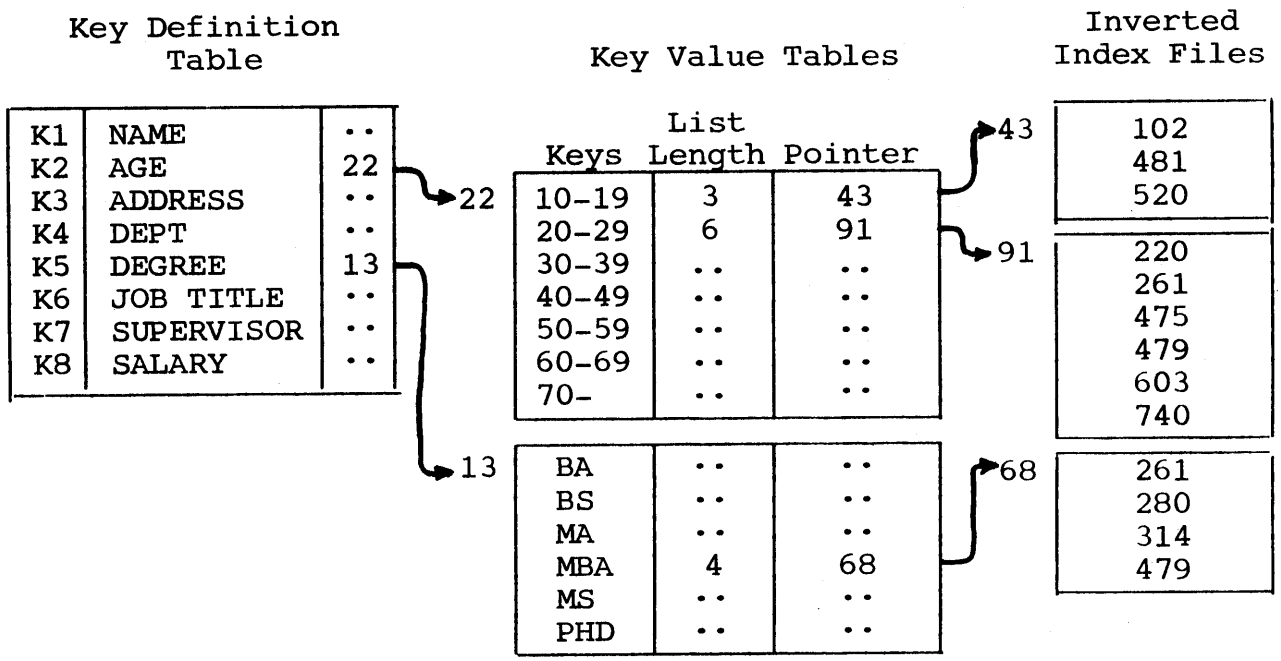


Figure 2: The Inverted Index File Organization

The inverted key lists are variable length records that must either be sorted prior to each use or be maintained in an ordered sequence for efficient logical manipulation. Both the maintenance of the lists as variable length records, which can

Figure 3: Example of the Handling of a Query by the Inverted Index System

QUERY — PRINT NAME WHERE DEGREE EQ MBA AND AGE LE 29



Note that the Key Value Tables can break down certain key fields into ranges of key values as well as into individual key values. After performing the appropriate list merging and intersections in core, the system identified two records that satisfy the query; they are located at addresses 261 and 479.

261

K1	SMITH, J.
K2	25
K3	BOSTON, MASS.
K5	MBA
K5	PHD
K8	15000
K7	RAMSEY, S.

479

K1	JONES, L.
K2	29
K4	7144
K5	MBA
K8	13400

be quite diverse in size, and their maintenance in sequence, contribute to certain programming complexities that are absent from the Multilist organization, although they buy much in performance.

Note that whereas the Multilist organization distributes the link addresses throughout the file, the Inverted Index collects and compacts the address pointers into a single area (the key directory) outside of the file. Thus, although the directory of the Inverted Index organization is considerably larger than that of the Multilist system, the total memory usage is no greater since these same address pointers no longer appear within the record. In fact, the Inverted Index system may require less storage than the Multilist system if the key names or codes are not individually cited within the record itself. This can be done if it is not necessary to print the keys as output; however, in so doing, there is no easy method of chaining a single record back to others associated with it.

As mentioned previously, the principal advantages of the Inverted Index organization are its general efficiency in retrieval operations (since it accesses only those records it needs) and its accurate pre-retrieval statistic that allows the user to modify his request based on the knowledge of the size of the response. Disadvantages of the Inverted Index system include the following:

(1) It is more complex to program than the Multilist system.

(2) It generally is less efficient than Multilist

in adding and deleting records; both cases require the modification of each inverted index that corresponds to a key in the newly added or deleted record.

(3) It requires a working area in order to perform the logic processing (list intersection, merging, etc.).

(4) Since the list records are variable length, some reserve is needed in them if real-time update is to be allowed.

2.3 The Controlled List Length Multilist

Lefkovitz has proposed a file organization that is a compromise between Multilist and Inverted Index. Under this file organization scheme, the Multilist system is modified by restricting each list to a predetermined maximum length. When a list would otherwise exceed the maximum length, a new list is started and the address of the first record on this new list is also put in the key directory. This system not only prevents lists from becoming too long to process efficiently, but also permits the data base to make use of the operating system's capability of overlapping I/O requests so as to reduce the overall list accession time.

For example, in Figure 4 the maximum list length is set to 4. Thus, list W, which contains seven records, is broken into a list of length four and a list of length three. The first list begins in Cell 0 at record six and is identified as A0.6; the

second list begins in Cell 1 at record nine (A1.9). Schematically, the same thread that appeared in Figure 1 is shown in Figure 4; however, the break in the single list is indicated by the broken line since, in fact, the seven items on the W list are now contained on two separate lists.

The search for records containing keys W AND X in Figure 4 begins by accessing the list of lists associated with key W and key X. It is found that there are two lists under each key; however, those under key X are shorter. Therefore, the lists beginning with records A0.3 in Cell 0 and A1.9 in Cell 1 are searched. Two records are found on these lists which are also on the lists for key W, and these are the responses to the query.

Moreover, it is possible to overlap disk accessions if two or more list records that are the "next to be accessed" are contained in different disk modules. If the cells in Figure 3 are defined as individual disk modules, then since the two sublists for key X begin in two different modules, the head positioning for records A0.3 and A1.9 could begin simultaneously. Hence, the two lists would effectively be searched in parallel, thus reducing the number of random accessions needed from 6 to 4.

A query of the type "X OR Y" necessitates the retrieval of both the X and Y lists. This allows record accessions from three sublists to be overlapped, with accessions in modules 0, 1, and 2 which in turn permits a high degree of parallel accession without any special programming, since the stacking of I/O commands is performed automatically by most manufacturer-

provided operating systems. Of course, the overlapping of disk accessions could be performed in the Inverted Index system too.

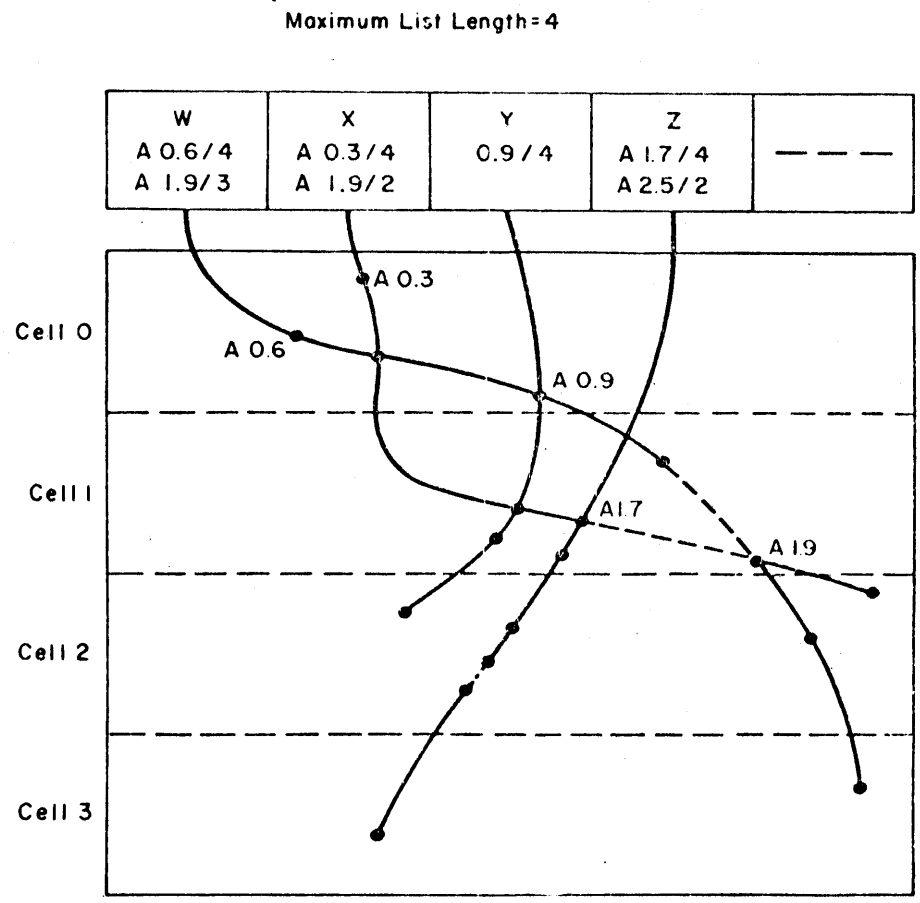


Figure 4: The Controlled List Length Multilist

The Multilist system with controlled list lengths can be viewed as being in the middle of a list structuring spectrum, the two extremes of which are the pure Multilist and pure Inverted Index: the Multilist has a list control of infinity; the Inverted Index has a list control of one. Moreover, the list control should be parameterized so that the file can be regenerated at any time with a new list length control, and hence a different degree of inversion. An alternative to the

single parameter is a table that provides differential control over the various keys. Frequently used keys should have shorter lists (i.e., more inversion) than infrequently used keys. Furthermore, this table could automatically be modified by a program that maintains key usage statistics, and the file reorganized periodically in accordance with the new state of the table. Such a procedure would have a self-adaptive quality.

Unfortunately, the presearch statistic in this file structure is still the same as that of the Multilist organization because an entire list (the shortest in the conjunction) must still be searched, and thus no intersections are possible at the directory stage of the search. Moreover, this organization is substantially more difficult to program than either Multilist or Inverted Index, and it generally needs more time to update. The search time will always be less than with Multilist and greater than that achievable with a pure Inverted Index, but it can be improved over time by means of the adaptive procedure described above.

2.4 The Cellular Multilist File Organization

The next step is to partition the data file and restrict list lengths on the basis of partition boundaries rather than length. A file organization of this type is called a Cellular Multilist File (Figure 5). The file partitions are usually based on some direct access device module such as a disk track or cylinder. The entries in the key directory contain a head of list address, the list length for each list, and a code indicating which cell the list is in. In searching this type

of a file structure, often it is not necessary to search all lists associated with a key. Some presearch processing can be done to determine which of the search keys involve lists in the same cell, and only those cells need be processed. There is also an advantage of being able to read a whole list from a direct access storage device without any physical movement on the part of the access mechanism if the partitions are defined correctly.

Consider the query X ∧ Z. An examination of the directory shows that list X contains sublists that are wholly contained within Cells 0, 1, and 2. Similarly, list Z has sublists that are entirely contained within Cells 1, 2, and 3. Since list X contains a head of list address in Cell 0 but list Z does not, no intersection of X and Z exists in Cell 0. Similar reasoning applies to Cell 3. Therefore, the search on the conjunction X ∧ Z would be limited to those sublists of X and/or Z contained only in Cells 1 and 2; furthermore, in Cell 1 list Z is preferred because it has a list length of 2 versus 3 for list X, and in Cell 2 list X is preferred with a list length of 1. The total list search for the X ∧ Z conjunction is then 3, instead of 6, as it would be if the shorter of the two lists X versus Z were searched, as in the Multilist file organization. A similar approach can also be applied to the Inverted Index method, whereby the inverted lists for a given cell would appear at the beginning of the cell.

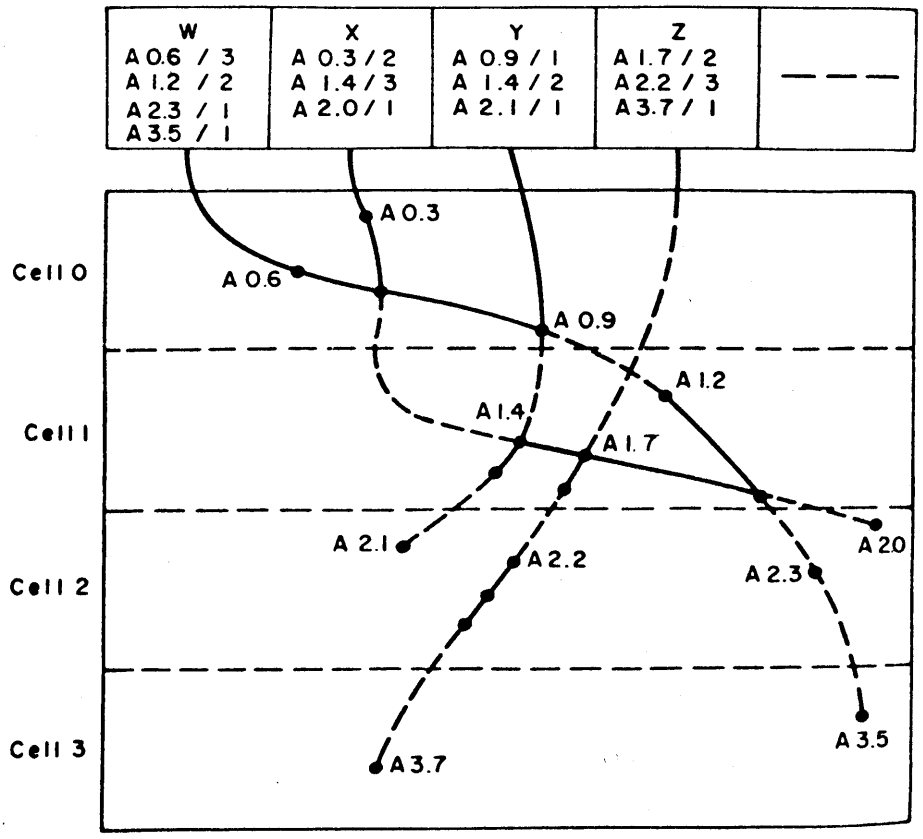


Figure 5: The Cellular Multilist File Organization

In comparison with Multilist, the Cellular Multilist reduces the total number of list searches and reduces head positioning time by keeping most of its list searches intracell. Although extra retrieval time is needed to access the cellular sublists, the overall retrieval time of this organization should still be lower than that of the standard Multilist. However, Cellular Multilist's updating time will be worse and its programming complexity is considerably greater. On the other hand, the retrieval speed of the Cellular Multilist will generally not

be as good as with the Inverted Index organization because it is still retrieving some records that do not satisfy the query.

One strategy for implementing the Cellular Multilist file organization calls for each cell to be on separate disk modules, thus allowing for a controlled overlap in the searching of different modules. Another alternative that is particularly appropriate for movable head disks is to equate the cells to disk cylinders, so that once the heads have been positioned to a given cylinder, the sublist search can be effected without any further head motion.

The assignment of cells to cylinders facilitates an effective strategy in multi-terminal, real-time systems: the individual cell accession lists from different user queries are merged together and the access heads are moved sequentially across the cylinders intermixing the list searches of several queries. When the head reaches the highest cylinder, the process is reversed. Mean access time is reduced both because the list searching is within a cylinder and because the cylinder head motion is more localized. The system could also be constructed to allow high priority queries to force the heads to jump to the appropriate cylinders.

2.5 The Cellular Serial File Organization

The next step along the progression is to eliminate the list structures and process the partitions serially. This organization is called a Cellular Serial File (Figure 6). The entries in a key directory are associated with lists of the cells which

contain records indexed under that key. When a search is to be performed, presearch processing can be done to determine which cells need to be examined. These cells are then examined serially. Again, the device dependent partitioning of a data set plays an important role in the efficiency of this method.

The query on keys X AND Z in Figure 6 begins by accessing the lists of cells associated with those keys. It is found that records associated with key X are contained in Cells 0, 1, and 2. Records associated with key Z are contained in Cells 1, 2, and 3. Therefore, Cells 1 and 2 are read serially and records which are associated with key X and key Z are retrieved as responses to the query.

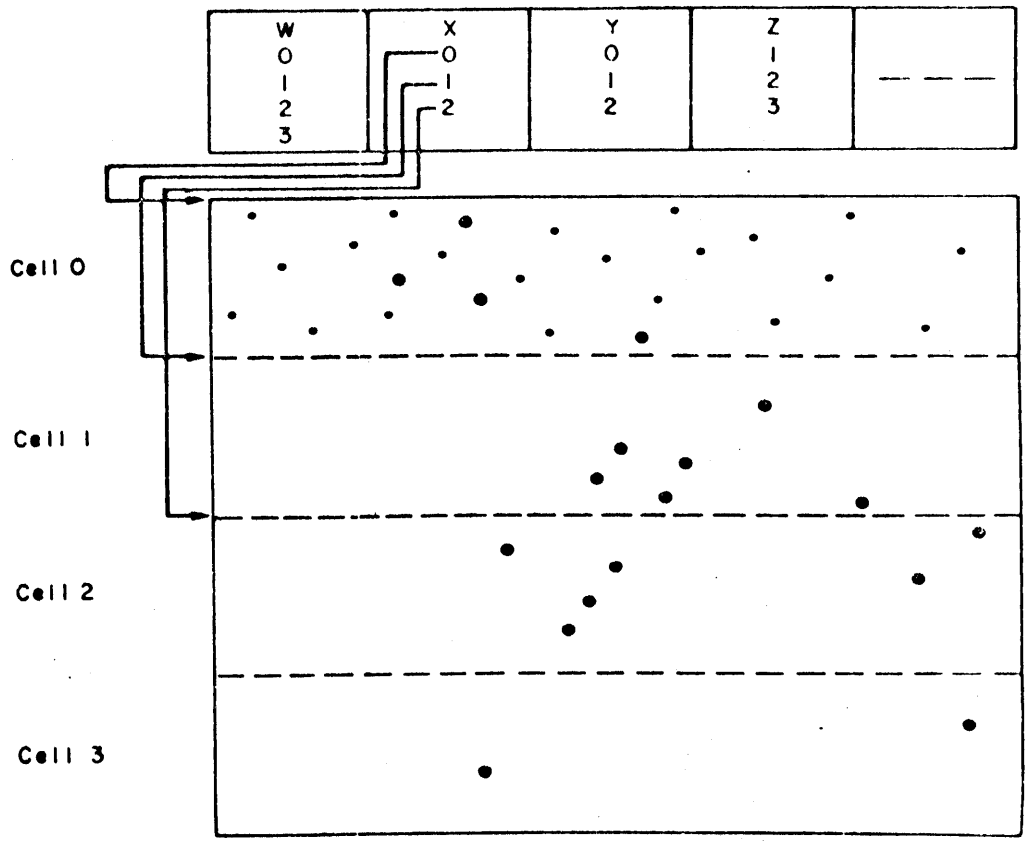


Figure 6: The Cellular Serial File Organization

The Cellular Serial approach has the advantage of programming simplicity and ease of update if reserve space is left at the end of the cell. It is applied with particular advantage to storage devices in which the average random accession time is high but the serial data transmission rate is also high. Unfortunately (for this method) at present, most of the bulk storage devices such as the IBM Data Cell that have a high access time of several hundred seconds also have relatively low serial data transmission rates of around 50 to 70 kilo-characters/second.

It is obvious that the fewer cells in which a given key is represented, the more efficient is the retrieval process for that key in the various cellular file organizations. In fact, for any of the multi-key retrieval organizations the average retrieval time is decreased whenever records having similar keys can be grouped together. The task of forming clusters of records that are frequently retrieved together is not a trivial one because any given record may have several keys in common with thousands of different records (although the specific keys they have in common may differ). Moreover, the exact statistical profile of the queries (e.g., which keys are often requested together) is usually not known before construction of the data base; and even when it is known, the best way of using the information is not immediately obvious. The entire topic of record clustering under multi-key retrieval is thus at the forefront of today's research; it will be taken up again in

Chapter VII. As for this paper, the retrieval analyses will be performed pessimistically, assuming that all records required to satisfy a given query are distributed randomly throughout the file.

CHAPTER III

AN OVERVIEW OF THE LITERATURE

It would be desirable to be able to make a clear distinction between the file organization techniques that can solely be used for single key retrieval and those that may be used for multi-key retrieval. This would allow a paper such as this that purports to treat only the latter, to ignore the former. Unfortunately, a clean break is not possible. Each of the individual files of the multi-key organizations can itself be organized in dozens of different ways (including hybrids and minor variations) — and each one of these ways is generally based on one or more of the single key structures.

For example, the data base files themselves under the Inverted Index organization can be organized either as a direct access file based on hash coding a unique Record Identity Number (RIN), as an indexed sequential file using the RIN as the primary key, as some form of hierarchical file, or as a simple sequential file with new records being added on at the end and with a separate inverted index organized on RIN used to locate each record. A similar file organization problem exists within each record of the direct file — particularly if hierarchical relationships are to be maintained — and for virtually every other type of file used in multi-key retrieval. It turns out that any multi-key file organization one can think of is usually just an integration of several single key methods, or can be viewed as such.

This paper assumes that the reader is familiar with all of the standard single key retrieval techniques; it does not require that he know all the possible variations in intimate detail. However, for those readers who do not have the necessary background, or for those who wish to delve more deeply into some of the single key techniques, the following section is provided. Its intent is to provide a brief, but fairly complete review of the literature, thus allowing a reader at virtually any level to augment the state of his knowledge.

3.1 The Single Key Retrieval Techniques

As is typical in the information retrieval area, there is a relative paucity of good overview articles that introduce the reader to several single key retrieval techniques and that suggest when each one might be appropriate. Price (70) in a 1971 article in the ACM Computing Surveys provides one such article; Dodd (27) in a 1969 article in the same journal another. Both articles are quite readable and have good bibliographies. The Dodd article is particularly good in that it shows how the basic single key techniques may be integrated to provide retrieval on multiple keys. A 1969 paper by Chapin (15) does not appear to be as useful for the novice because of its non-standard terminology and its lack of good diagrams and illustrative examples; however, he too provides a fairly good bibliography.

An almost-must is IBM's Introduction to IBM System/360 Direct Access Storage Devices and Organization Methods (41). It is well written, covers both direct access hardware and IBM-supported single key retrieval techniques, and contains a number

of useful student exercises with supplied answers. Perhaps the best introductory book in the field has been written by Ivan Flores (28). His coverage is complete and readable, but he gives no bibliography. Finally, extensive annotated bibliographies on all types of information retrieval techniques can be found in several volumes of the Annual Review of Information Science and Technology (18,61,82,80,59). The relevant chapters are highlighted under the multi-key section of this chapter.

The straightforward sequential file is not covered here because its properties have been reviewed numerous time elsewhere. The binary search technique is discussed in the Price article (70) and the Flores book (28). Flores and Madpis (29) devised a formula for the average number of "looks" to find a given record in an arbitrary-sized table. Collmeyer and Shemer (22) compared the performance of sequential searching vs. binary searching for files stored on disk. Rothnie (76) as part of a recent Ph.D. thesis examined the benefits of binary searching in a paged environment, and concluded that it performs poorly relative to such techniques as directories or hash coding because it scatters its memory accesses over many pages.

The area of direct access techniques (also called random access or hash coding) is blessed with several excellent articles. Peterson's 1957 article (69) is still a classic in the field. He simulated the results of the open addressing systems using buckets of various sizes, compared the simulated performance with the performance of four actual business record files, and then derived theoretical results for the single bucket case.

Buchholz (12) in 1963 and Morris (62) in 1968 wrote excellent review articles summarizing the existing key-to-address transformation methods as well as the techniques for handling key collisions.

Maurer (58) in 1968 proposed a division hash code with quadratic residues used to handle collisions. Several articles have extended Maurer's technique, notably Radke (74), Bell (5), Lamport (47), and Day (25). In 1971, Lum et alii (56) undertook a fairly exhaustive experimental evaluation of several of the existing key-to-address transform techniques, varying both the load factor and the bucket size over a wide range: their results showed the division technique to give the best overall performance. Since that time, Bell and Kaman (6), Luccio (52), and Brent (11) have proposed new methods for which they claim even better results. Recently Lum (53) has developed an analytic approach to the analysis of key-to-address transformations. It is based on the behavior of the transformation in the key space and on an abstractly defined set of sample files. This work has substantiated Lum's earlier result that the division method performs best, and has explained this result analytically.

Bays (4) and Knott (45) have considered the problem of expanding or reallocating hash-coded tables, and Nijssen (65) has demonstrated how, contrary to previous opinion, updates to random access files can be efficiently performed in the batch mode.

Because they are so much a part of the popular indexed sequential access method, hierarchical directories are not often

treated as a separate issue. Collmeyer and Shemer (22), who term them "tabular indices", found that they performed favorably in a disk file environment, and Rothnie (76) demonstrated that they do well in a paging environment. Landauer (48) implicitly considered them within the context of balanced trees, and Lefkovitz (49) analyzed them as one of several alternate techniques available for directory decoding.

The indexed sequential organization itself has been well studied in several different papers. The IBM Student Text (41) and the Flores book (29) both provide good introductions to it. Lum, Ling, and Senko (55) reported on the results of a complex simulation study they performed on the indexed sequential access method (ISAM) using their FOREM program. This program allowed them to simulate thousands of different ISAM data bases and study retrieval times as a function of such parameters as number and placement of index levels, overflow configurations, types of transactions, and percent of records overflowing.

Several other papers — notably those by Ghosh and Ganguly (34), by Coyle (23), and by Mullin (63) — have suggested changes in indexed sequential to improve its overall performance. An alternative indexed sequential system has been developed called "AMIGOS" (89) that is not quite as flexible as ISAM but that claims to be nearly ten times more efficient.

Chained or linked list structures are covered in many places. Dodd (27), Flores (28), and Stone (86) all provide good introductions. Perhaps the most important single reference in this area is Volume 1 of Knuth's The Art of Computer

Programming (46). Knuth's presentation of single and doubly-linked lists, circular lists, garbage collection, and dynamic storage allocation is not only thorough, but it contains numerous exercises at various levels of difficulty that can be performed by the student. Gray's 1967 paper (35) summarized several documented varieties of ring-structured files (i.e., circular lists), and D'Imperio (26) provided a detailed review in 1969 of the principal list structuring and string manipulation systems that had been developed. In a paper more closely related to information retrieval, Jones (43) reported on an actual implementation of a large data base using the IDS list-structed system.

Tree structures are closely related to lists because of their heavy usage of pointers. One of the key articles on tree structures was written by Sussenguth (87) in 1963. He demonstrated that a doubly-chained tree structure would permit an efficient compromise between the fast search/slow update characteristics of binary search and slow search/fast update characteristics of linked list structures. Patt (67) relaxed the limitation by Sussenguth that all terminal nodes had to lie at the same tree level, and developed a procedure for constructing trees with minimum average search times. Other authors who have studied the properties of various types of trees include Hibbard (38), Clampett (17), Arora and Dent (2), Knott (44), Foster (30,31), Knuth (46), Landauer (48), Rotwitt and deMaine (78), Skidmore and Weinberg (84), and Coffman and Eve (21). Casey (14) has written a recent paper describing

an approach for minimizing the number of nodes tested in a tree-structured file used for multi-key retrieval. The procedure he describes consists of organizing the tree nodes based on ORing records such that records that satisfy a sampled set of typical queries are grouped together within the tree.

3.2 The Multiple Key Retrieval Techniques — Descriptions

Once again a basic problem is the relative lack of good tutorial papers on the subject. There seems to be only three: the previously mentioned papers by Dodd (27) and Chapin (15), and a 1973 paper by Cardenas (13). Dodd's paper is useful because of his clear diagrams and descriptions of how the complex multi-key file organizations are implemented by combining the basic single key data structures. The Cardenas paper is a must for any potential file designer, because in a few short pages he discusses a great many of the factors that affect file performance and cogently summarizes most of the relevant considerations in selecting a file organization. Moreover, he performs a valuable service by specifically proposing that the best file structure in a given situation is one that minimizes a cost function that includes retrieval, update, storage, programming, maintenance, and other unquantifiable costs, while satisfying given design constraints (e.g., query response time must be less than three seconds for 90% of queries). Cardenas and Dodd both provide good bibliographies.

Another source of papers in the multi-key area is the Annual Review of Information Science and Technology. Each of the first five volumes of this excellent series contains an entire chapter devoted to reviewing many of the articles and books that appeared during the previous year on the subject of file organization. The relevant chapters are located in the 1966 through 1970 volumes and were written by Climenson (18), Minker and Sable (61), Shoffner (82), Senko (80), and Meadow and Meadow (59). The articles by Senko and by Minker and Sable are to be particularly recommended.

The two CODASYL Systems Committee reports on generalized data base management systems — the 1969 Survey and the 1971 Feature Analysis — shed some light on the file organization approaches of these generalized systems. However, the primary thrust of the two reports is in analyzing the user-oriented features available in these systems, not their on-line performance or their internal file organizations.

With respect to the two basic file organizations discussed earlier in the paper, it is a moot point which of them is older. The Multilist technique dates back to the late 1950's and early 1960's. Perlis and Thornton (68) and Weizenbaum (93) foreshadowed it with their "threaded lists" and "knotted lists" respectively, and in 1962 Prywes and Gray (72,73) described a more general formulation called multiple threaded lists or Multilist. Since that time, several interesting applications using Multilist have appeared in the literature including implementations by Prywes (71), Hsiao and Prywes (40),

Wexelblat and Freedman (94), and Lefkovitz and Powers (50) — the last of which describes an application involving the Cellular Multilist system.

The Inverted Index systems seem to have been adapted from the indexing techniques used in libraries. An early article by Johnson (42) delineated the basic methodology, and applications involving the Inverted Index organization have been described in papers by Bloom (9), Weinberg (92), O'Connell (66), and Davis and Lin (24). The O'Connell and the Davis and Lin systems are both simple in that they operate on smaller data bases and do not allow an internal hierarchical structure; however, both are interesting. O'Connell's system was implemented at a cost of under $10,000 by using only the IBM-supported Indexed Sequential and Direct Access Methods. Davis and Lin made use of a clever and efficient bit-mapping scheme to achieve the logical intersection of attribute lists. They also included both theoretical and actual figures on the time needed to perform various types of retrieval requests using their system.

One of the most sophisticated Inverted Index systems ever implemented was the Time Shared Data Management System (TDMS) developed by the Systems Development Corporation and described in published papers by Franks (32), Williams and Bartram (95), Bleier (7), and Bleier and Vorhaus (8). A TDMS data base was almost fully inverted, with each attribute being indexed unless the user specifically requested that it not be. The Bleier and Vorhaus paper describes the overall TDMS file structure;

the earlier paper by Bleier shows how TDMS handled hierarchical data associations within the data base records. Siler (83) has stated that TDMS and its follow-on CDMS (Commercial Data Management System) were most likely too machine dependent and too inverted. Because of its total inversion of the data base and its complex hierarchical structure, CDMS exploded the size of original data bases by a factor of from two to five. The Systems Development Corporation now offers an alternative system called DS-1 (88) which allows the user to specify which attributes he desires to have inverted.

3.3 The Multiple Key Retrieval Techniques — Comparative Analyses

Several papers have appeared in the literature, and at least a couple of Ph.D. theses have been written comparing the Multilist and Inverted Index (and occasionally other systems) along various dimensions. A 1963 paper by F. T. Baker (3) that is part of a series of information retrieval reports originating at Harvard contains one of the earliest comparison studies. Baker compared Multilist, Inverted Index, and sequential files with chaining on secondary keys. Although his formulas were not totally correct, his results clearly showed several facts:

(1) that Inverted Index performs better the higher the proportion of retrievals to total system operations, while Multilist performs worse;

(2) that Multilist does worse the greater the number of records that contain a given key value;

(3) that Multilist performs at its best the simpler

the queries, while Inverted Index performs the same regardless of the complexity of the queries. (This latter statement is probably not true.)

Overall, the various file organizations exhibited mixed successes with no one technique being best in all cases.

T. C. Lowe (51) attempted to derive an analytical model of memory utilization and retrieval time for the Inverted Index and Multilist file organizations. Lowe's work is particularly interesting in that he included two important distributions in his analysis: f(j), the number of times the j^{th} unique index term is referred to within the data base; and p(j), the probability that the j^{th} unique index item is used in a query. Lowe performed three analyses for each file organization, assuming that f(j) and p(j) were both uniformly distributed; that p(j) was uniform and f(j) was distributed according to Zipf's law;[4] and that both had Zipf distributions.

Unfortunately, Lowe's work is difficult to apply to actual data bases principally because he does not distinguish between simple and complex queries. This penalizes the Inverted Index technique since its strength lies in its ability to access only those records that are "fully" qualified (depending on the

[4]The Zipf distribution for p(j) is $p(j) = C/j$ for j=1 to N where C is a constant such that $\sum_{j=1}^{N} p(j) = 1$

hybrid of the Inverted Index system) for the query. In addition, in real world situations the distributions of f(j) and p(j) may be unknown or difficult to determine. Siler (83) in one of his footnotes points out other limitations of Lowe's analysis.

In 1969, David Lefkovitz (49) wrote File Structure for On-Line Systems, which even today is the most comprehensive, design-oriented reference on multi-key retrieval systems. This highly readable book covers several important facets of the problem:

(1) It summarizes the major software components required in a real-time information retrieval system.

(2) It presents an outside-in view of the system by means of a prototype query language (data manipulation language).

(3) It discusses various techniques for the design of directories and their decoders.

(4) It describes in detail several multi-key file organization techniques and the procedures they employ for data retrieval and updating.

(5) It presents accurate timing formulations for both the retrieval and updating operations (on which the timing formulas of this paper are largely based).

Lefkovitz's clear discussion of the techniques for performing the various kinds of file updating on Multilist and Inverted

Index data bases appears to be almost unique in the literature. Unfortunately, Lefkovitz seems to have made a few errors in his timing analyses and in calculating some of his examples.

Another important paper in the area of comparing system performance was written by L. D. Martin (59). Martin's paper leans heavily on the work of Lefkovitz (as we all do), but using a more approximate timing analysis, he suggests a comparison model based on minimizing the average on-line processing time, defined as being = (p)(retrieval time) + (1-p)(update time). However, once he derives his approximations for processing time, he then spends three full pages contorting the formulas so that they can be compared bilaterally. Martin's resulting model is not only non-intuitive, but it produces output numbers that are totally divorced from a physical interpretation, and in certain circumstances it cannot even identify the "best" organization.

Furthermore, Martin appears to have made two critical errors in his test cases of the model that negate much of his sample results. These suspected errors, along with those of Lefkovitz, are delineated in Appendix B of this thesis, precisely because these two papers are otherwise so critical to the development of the field.

Hsiao and Harary (39) have written a paper that describes a generalized file structure and shows how some frequently used file organizations are special cases of their general structure. They show, for example, that whereas the Multilist organization has only one list per keyword stringing together

all N records containing that keyword, the Inverted Index organization has N lists per keyword each of which contains just one record. The General Retrieval Algorithm they develop proves useful with the Multilist organization in avoiding multiple retrievals of the same record when answering queries involving an OR operation.

The 1973 paper by Cardenas (13) described earlier discusses a number of factors that affect file organization performance, and then describes a simulation model developed at U.C.L.A. that compares three file structures — the Inverted Index, the Multilist, and the doubly-chained tree. Six different data bases were structured within the simulation according to each of the three file techniques, and then results were collected on total storage requirements and on average retrieval times for queries of various complexity. The storage requirements of the Multilist and the Inverted Index organizations were usually very close, while the requirements of the doubly-chained tree were generally somewhat less. The results of the retrieval timing simulation showed specific instances where each file organization did significantly worse than the other two.

Cardenas' results are less useful than they might otherwise have been because the six data bases he used appear to have been overly similar in terms of structural characteristics (e.g., number of records, number of keys, etc.), and thus not truly representative of the wide spectrum of possible data bases. Moreover, Cardenas did not model record updating within his

simulation, present the retrieval timing formulas he used, nor try to assess what were the characteristics of each of the data bases that made it perform better or worse under the given file organization techniques. However, these limitations do not detract from the modelling methodology he presented nor from his excellent discussion of the relevant factors that affect data base performance.

Cardenas seems to have based some of his work on two theses done at U.C.L.A. — a 1971 Masters thesis by J. P. Sagamang (79) and a 1972 Ph.D. thesis by K. F. Siler (83). Siler developed a generalized simulation model to study the efficiency of three different file organizations — the Multilist system, the Inverted Index system, and the Cellular Multilist system — and found that Multilist was quickest for very simple queries, while Inverted Index was most efficient for more complex queries. Siler also proposed and simulated an "Integrated List" system, in which each key could be organized under any of the three above file organizations. He described how complex queries involving keys organized under different schemes could be handled by the integrated system, and concluded that given the varying effectiveness of the three studied organizations under different query schemes, a data retrieval scheme involving mixed file organizations could be effective enough to offset the increased programming problems. Finally, Siler performed a number of simulation runs which showed that key aggregation (i.e., $30 \leq AGE \leq 39$) becomes increasingly effective as the query range increases so long as the level of key aggregation is no

greater than the range of the queries.

Siler himself readily conceded the limitations of his work and the need for additional research. Record updating was not considered at all, nor was any attempt made to study the effects of record clustering. Siler also recommended development of a dynamic time simulation, as opposed to a static one, to answer the time-related problems of handling multiple on-line users simultaneously. Hopefully, additional work will also be done in the fertile area of static simulation to draw other general conclusions concerning retrieval and updating efficiency, and to study further Siler's proposed Integrated List system.

Another approach to the problem was taken by A. J. Winkler (96) in his 1970 Ph.D. thesis at the University of Texas, and in a follow-on paper he wrote with A. G. Dale (97). Following the direction taken by Martin (57), Winkler developed detailed timing formulas for four different file organizations, which could then be used to choose the organization that would minimize the expected processing time. The four structures chosen were the doubly-chained tree, the triply-chained tree, and two Inverted Index systems involving hierarchically-organized data records. In the first Inverted Index system the pointers linked directly to the node within the hierarchical record, while in the second they linked to the start of the record (and thus if the same key appeared more than once in any record, there was only one pointer).

Because of the complexity of the four file organizations, Winkler's processing algorithms are exceedingly detailed and complex. Moreover, his timing formulas include in-core processing as well as disk accesses and thus they too are quite involved. His results for overall processing under various parameter values showed that the two Inverted Index systems dominated the two tree structures, although the choice between the two Inverted Index systems was not conclusive. The two Inverted Index variations will be treated in more detail in Chapter VI, along with other options available in structuring such a system; the two tree-structured systems will not be discussed further since Winkler showed that for typical retrieval requests they required approximately six disk accessions (at .2 seconds each) for each file record, leading to lower bound of 120 seconds even for the case when not one record out of a 1000-record data base satisfied the query.

CHAPTER IV

DEVELOPMENT OF RETRIEVAL TIMING FORMULAS

This chapter is concerned with developing a parameterized model for estimating the retrieval times of three of the file organizations discussed in Chapter II given inputted values for the data base, query, and storage device characteristics. The three organizations selected for this analysis are the Multilist, the Inverted Index, and the Cellular Serial. The principal reasons for selecting these three organizations were that they represent three clearly different approaches to the data retrieval problem and the fact that some analytical work on retrieval timing has already been done for these organizations, largely due to the work of Baker (3), Martin (57), Lefkovitz (50), Siler (83), and Winkler (96,97). No specific formulas are developed for the retrieval or updating time of the other two organizations (Controlled List Length Multilist and Cellular Multilist), since such formulas would ultimately depend on making special assumptions for characteristics (particularly the amount of I/O overlapping that can be done for a typical query) that have no direct parallels in the three analyzed file organizations.

Once the basic retrieval timing models have been developed, a number of "reasonable" examples will be presented illustrating the use of the model in typical data base situations. The device-related characteristics for these examples will be based

on two representative (and popular) direct access storage devices: the IBM 3330 movable head disk and the IBM 2321 Data Cell. The 3330 is fast but expensive; it can retrieve a 1000-byte record in about 35 milliseconds. The data cell is a forerunner of a number of slow, cheap, bulk storage devices just now coming on the market; it can retrieve a 1000-byte record in around 500 to 600 milliseconds.

4.1 Description of Representative Storage Devices

Each IBM 3330 Disk Pack consists of ten disks with nineteen surfaces used for recording (the top surface of the top disk is not used for recording). The disks are accessed by a comb-type mechanism having nineteen read/write heads — one for each recording surface. On each disk surface there are 404 tracks and seven alternates. The overall disk pack can hold one hundred million characters (bytes); a disk control unit with eight disk packs can thus hold up to 800 million characters.

The time required to access and transfer data on the 3330 consists of three parts: head positioning time, rotational delay (latency), and data transfer. The head positioning time is the radial motion of the heads across the disk to the appropriate track. Since the access heads all move to the same track together, the 3330 disk is often viewed conceptually as being comprised of 404 vertical cylinders, each containing 19 tracks. Thus, if the mechanism is already in the correct cylinder there is no need to move it, so the positioning time is zero. The rotational delay is the time required for the correct data to

rotate to the read/write head so that data transfer can begin. Since this latency time can range from zero to a full revolution, half a rotation (average rotational delay) is generally used for timing purposes.

The IBM 2321 Data Cell Drive consists of 10 data cells, each of which can hold 200 magnetic strips. Each strip holds 100 tracks, allowing a maximum capacity of 200,000 bytes per strip. The array of 10 data cells is rotated in either direction until the strip to be processed is under a drum that is fixed in position above the array. The drum then rotates to pick up the strip and move it past a bar of 20 read/write heads. The heads are positioned at each fifth track of the strip. The bar of heads can move horizontally to five different positions, thus providing access to all 100 tracks of a strip. Since there are 20 heads and 100 tracks per strip, each magnetic strip can be conceptually viewed as containing five cylinders of 20 tracks each.

The access or positioning time may include restoring the strip on the drum and/or picking up a new one. It takes 200 ms to restore a strip, from 75 to 225 ms to rotate the array, and 175 ms to pick a strip. The movement of the bar of read/write heads takes 95 ms; this is overlapped with the restore of pick if either occurs. The minimum access time is 95 ms if the correct strip is already on the drum and only head bar motion is required; if the same cylinder within the strip is being accessed, the positioning time is zero. The maximum is 600 ms: 200 ms to restore, 225 ms to rotate, and 175 ms to pick. The rotation of

the drum takes 50 ms and thus the latency delay is approximately 25 ms.

Table 1: Device Storage Summary

Device	Usable Cylinders	Tracks per Cylinder	Bytes per Track	Bytes per Cylinder	Bytes per Unit (Million)
3330	404/pack 3232/unit	19	13,030	247,570	Pack: 100 Unit: 800
2321	5/strip 10,000/unit	20	2,000	40,000*	Unit: 400

*200,000 Bytes/Strip since there are 5 Cylinders/Strip

Table 2: Device Timing Summary

Device	Positioning (ms)	Latency (ms)	Rotation (ms)	Data Transmission Rate (Kilo-Bytes/Sec)
3330	27	8.4	16.7	806
2321	350/550*	25	50	55

*350 assuming that the previous strip already restored;
550 otherwise

4.2 Key Directory Decoding Time

One of the most important files in any information retrieval system is the key directory. For each key specified in a query the key directory locates the proper directory record, which in turn either points to the inverted list for that key (in the case of the Inverted Index organization) or the first record in the data base containing that key (in the case of the

Multilist organization). Locating the proper directory record efficiently in a directory containing thousands or even millions of unique keys is not a trivial task, and several different techniques for decoding directories have been developed. A few of these will be discussed briefly in Chapter VI. However, for our current purposes we have to assume some specific structure for the key directory in order to develop the overall timing analyses. In the case of retrieval operations, it does not matter too much which directory decoding technique is assumed, since the directory decoding time affects all three file organizations about equally, and moreover, it is usually dominated by the time required to access the data base records. Directory decoding time is generally more critical for on-line updating operations, particularly record insertions and deletions in systems where the key lists for all keys in the record are immediately updated.

In this chapter and the succeeding one we shall assume that the directory decoder has been implemented using a three-level hierarchical index. This is probably the most popular directory decoding technique in use today. We shall assume that the first level of the hierarchical index is located in core and the second and third levels on a movable head disk, such as the IBM 3330. Furthermore, it will be assumed that each set of third-level nodes emanating from a given second-level node is stored in the same cylinder as the second-level node, so that there is no head positioning required between the second and third levels. The decoding time under these assumptions is as follows:

Table 3: Decoding Time for Three-Level Tree

Decoder Process	Tree Level	Time (Symbolic)	Time for IBM 3330 Movable Head Disk
Process Record in Core	1	0	0 ms
Head Position		P	27
Latency		½R	8.4
Track Read		½R	8.4
Process Record	2	0	0
Lost Revolution		R	16.7
Track Head		½R	8.4
Process Record	3	0	0
			Total 69 ms

The above formulation assumes that the specific key directory record desired is located halfway through a given track thus requiring 8.4 milliseconds = ½ x 16.7 ms (≈ ½ x 13KB/806KB/sec).

4.3 Retrieval Timing Formulas

The purpose of this section is to derive approximations for the average access time to satisfy a typical query for each of the three file organizations being considered. Timing formulas are generally a function of three different types of parameters:

(1) The relevant characteristics of the data base (e.g., average number of keys per record, total number of unique keys in the file, total number of records)

(2) The relevant characteristics of the query (e.g., the number of nonnegated keys in the query, the total number of records retrieved

in response to the query)

(3) The characteristics of the direct access storage device (e.g., its transfer rate, positioning time, storage space per track and per cylinder)

Table 4 contains a full listing of the relevant parameters and their definitions. Some of them are readily obtainable from manufacturers' specifications or file generation statistics; others can only be estimated to the best of the file designer's or the eventual users' abilities.

One approximation that is made consistently throughout this analysis is to ignore the time needed to perform in-core processing, such as table lookups, calculations, and bitmap processing. This seems reasonable since currently the time to access core memory is on the order of 10,000 to 1,000,000 times faster than the time to access external (disk or data cell) memory. In essence, the data retrieval systems are assumed to be totally input-output bound.

Mendelson (60) explains how all queries can be represented by either of two canonical forms: disjunctive normal form and conjunctive normal form. Disjunctive normal form consists of one or more disjuncts (OR's) each of which is a conjunction or product of one or more keys (e.g., $(A \wedge B) \vee (A \wedge C) \vee E$). Conjunctive normal form is the reverse of this. As previously mentioned, this paper assumes that all queries are transformed into disjunctive normal form. All of the query-related parameters in

the timing formulas are based upon a single query product (i.e., there are no OR's in the query). If the query is a sum of products, then the individual retrieval time for an average query product must be multiplied by the number of sums.

Table 4: Parameters Used in Retrieval Time Models

Symbol	Definition	Parameter Type
V	Number of distinct keys in the file	File Related
N_k	Number of keys/record (ave.)	"
N_r	Number of records in the file	"
L	Average list length ($\approx N_r N_k / V$)	"
C_f	Characters/record (ave.)	"
R_c	Records/cell (ave.)	"
C_k	Cells/key (ave.)	"
Q_t	Number of terms (i.e., keys) in a single query product (ave.)	Query Related
Q_n	Number of nonnegated terms in a single query product (ave.)	"
L_s	Shortest list length in query (ave.)	"
Q_r	Number of responses to the query (ave.) $(0 \leq Q_r \leq L_s)$	"
A	Number of file record addresses per DASD track	Device Related
P	Positioning time of DASD (ave.)	"
R	Rotation time of DASD (latency $\frac{1}{2}R$)	"
R_t	Transfer rate of DASD (kilo-chars/sec)	"
T_d	Time required to decode the directory (also a function of decoding technique)	"
T_a	Time required to access a record; $T_a = P + .5R + C_f/R_t$	"

The record access time (T_a) is the time to position the disk head (P) plus latency (.5R) plus the data transfer time

(C_f/R_t). The formulas for the retrieval time for the Multilist, Inverted Index, and Cellular Serial file structures are shown in Table 5.

The Multilist system decodes only the nonnegated keys of the query, and then, for the query key having the shortest list, it accesses every record on that list. The Multilist system performs no intersections of address prior to file search, and can start printing out answers as soon as it locates the first record on the list that satisfies all keys.

The Inverted Index system first decodes all keys in the query, and then accesses the address lists for each key, where each address list averages [L/A] tracks in length. ([X] denotes the integer rounding of X up to the next highest integer.) It is assumed here that all the inverted list tracks for a given key value are located on the same cylinder, so that only one head positioning movement is required. Inverted Index then performs the list intersections in core and retrieves only those records that satisfy all query conditions.[5] Thus, at the end of its list processing, and before accessing the records, the Inverted List system can tell the user how many records actually satisfy the query.

[5]For the IBM 3330 disk only [L/A] -1 tracks will need to be read in full; the last track of record addresses will generally be only partly filled. For a device like the IBM 2314 disk, which performs full track reads, the approximation is slightly more accurate.

Knowing this statistic, the user may change his mind about wanting to see them all.

The Cellular Serial system decodes only the nonnegated keys of the query, and accesses the list of cells containing each of the nonnegated keys, where each list of cells averages $[C_k/A]$ physical records in length. It then intersects the lists of cells, retrieving all cells having at least one record that satisfies the query conditions, and reads each intersected cell serially (where each cell contain R_cC_f bytes). Our present analysis assumes that the data base has not been specially constructed so as to group together records having similar keys. Under this assumption for a typically large data base "No. of Cells">>"No. of Records Desired", and the average cell would contain only one desired record. For this reason, the number of cells retrieved can be approximated by the number of records that satisfy the query.

Table 5: Retrieval Time Approximations

Process	Multilist	Inverted Index	Cellular Serial
Directory Decoding	Q_nT_d	Q_tT_d	Q_nT_d
Retrieval of Address Lists	—	$Q_t(P+1.5R[L/A])$	$Q_n(P+1.5R[C_k/A])$
Retrieval of Data Records	L_sT_a	Q_rT_a	$Q_r(P+\frac{R_cC_f}{R_t})$

Assuming that there are no negated terms in the average query (i.e., $Q_n=Q_t$), the directory decoding times for all three

file organizations are the same. Excluding the directory decoding time (which is usually relatively small anyway), the retrieval times for the three systems are as follows:

$$T_{ml} = L_s T_a$$

$$T_{ii} = Q_t(P + 1.5R[L/A]) + Q_r T_a$$

$$T_{cs} = Q_n(P + 1.5R[C_k/A]) + Q_r(P + \frac{R_c C_f}{R_t})$$

We can use these formulas to do some interesting comparisons. Assume that $T_a = P + 1.5R$, meaning that a full track is read each access, either because the records are large or a disk like the 2314 which reads full tracks is used, and that $[L/A] = 1$, meaning that all addresses for a given key are contained on one physical track. For the two most popular systems, Multilist and Inverted Index:

$$T_{ml} < T_{ii}$$

$$\text{if, and only if } \quad L_s < Q_t + Q_r$$

Taking some typical values, assume

$$Q_t = 3 \text{ terms}$$

$$L_s = 20$$

Thus, for the Multilist organization to be at least as efficient in this case as the Inverted Index, then $Q_r > 17$, implying that at least 85% ($=17/20=Q_r/L_s$) of the records on the shortest list would have to satisfy a three-term query.

This is highly unlikely since the typical range for Q_r/L_s would be in the range of .05 to .25.

4.4 Examples Using the Retrieval Model

A FORTRAN program has been written to calculate approximate record retrieval times based on the formulas developed in the previous section. The same program is used in Chapter V to run tests based on the fully developed model. A listing of the program is given in Appendix A.

Although the program allows the user to set all of the parameter values for the model, some experimentation has shown that given the basic size of the data base, the speed of the DASD, and the types of queries that are usually asked, three parameters turn out to be of particular importance:

L = Average list length for keys in query

L_s = Shortest list length in query (ave.)

Q_r = Number of responses to the query (ave.)

Thus, the program has been designed to run through a set of "test cases" in which these three parameters are varied while the remaining parameters are held constant. As can be seen in Table 4, the average list length L (the value of which is generally not known by most users) is closely related to three parameters that users may have a fairly accurate estimate of: N_r the total number of records, N_k the number of keys/record, and V the total number of unique keys. Because $L \approx N_r N_k/V$, L can be increased by a factor of 10 by increasing either N_r or N_k or their product by a factor of 10 or by decreasing V by a

factor of 10. For the sake of simplicity, N_k alone was chosen as the control variable, although it is L, which depends on N_k, that is the relevant variable. It turns out that in updating files N_k is the only one of the four variables that is relevant.

Two sets of thirteen test cases were run through the program. In the first set of cases the directories used by the three systems and the smaller inverted indexes for the Cellular Serial organization were assumed to reside on a fast random access storage medium (such as the IBM 3330 disk) and the data files and the larger inverted lists for the Inverted Index organization were assumed to reside on a cheap bulk storage device (such as the IBM 2321 Data Cell). In the second set of cases, all files — directories, inverted lists, and data base records — were assumed to reside on a fast disk. For both sets of test cases the fixed parameters were set at values that reflect typical large data base requirements; the assigned values are shown in Table 6.

A fairly typical large data base was defined for the test cases. It contains 500,000 records, each of which consists of 2000 characters, and it has 10,000 unique keys. The queries are assumed to consist of one query product (no OR's) having four terms, only one of which is negated.

On the 3330 disk a cell for the Cellular Serial organization was defined to be equal to a disk cylinder. Since each track can hold six records (13030/2000) and there are nineteen tracks/cylinder, a disk cell holds 114 records. On the 2321, a cell was defined to be one of the magnetic strips.

Table 6: Values of the Fixed Parameters

T_d = Time to decode the directory on 3330 disk = 69 ms

N_r = 500,000 records in the data base

V = 10,000 distinct keys

C_f = 2000 characters per logical file record

C_k = Minimum of (L , N_r/R_c)

Q_b = 4 terms per query

Q_n = 3 nonnegated terms per query

	3330 Disk	2321 Data Cell
Positioning Time — P	27 ms	500 ms
Rotation Time — R	16.7 ms	50 ms
Transfer Rate — R_t	806 KB/sec	55 KB/sec
Record Addresses per Track — A	1600	250
Record/Cell — R_c	114	100
Time to Access a Record — T_a ($P + .5R + C_f/R_t$)	38 ms	561 ms
Time to Read a Full Track ($P + 1.5R$)	52 ms	575 ms
Time to Read a Cell ($P + R_cC_f/R_t$)	310 ms	4136 ms

Each track on the strip thus holds one record, and a cell contains 100 records. Since the records are assumed to be dispersed randomly throughout the cells, the number of cells/keys

is assumed to be equal to the smaller of the average list length and the total number of cells (N_r/R_c). Because of some clustering, particularly as L approaches the total number of cells, this figure will always be slightly too large.[6]

Three values for N_k (the number of keys per record) were selected: 5, 20, and 80.

Note that

$$L = \frac{N_r N_k}{V} = \frac{500000 N_k}{10000} = 50 N_k$$

The other two variable parameters were set to several different values subject to the common sense restriction that for a query product

$$0 \leq Q_r \leq L_s \leq L$$

The results of the first set of cases in which the data files and inverted lists (under the Inverted Index system) are stored on the data cell are shown in Table 7. Table 7 clearly demonstrates the superior retrieval efficiency of the Inverted Index system — at least for this basic set of data base characteristics.

[6]Rothnie (76, pp. 103-110) develops a Markov model that can be used to find a better estimate of the number of cells that need be retrieved given the number of records per cell, the total number of cells, and the average list length. For example, in a file containing 64 cells with 100 records/cell the expected number of cells retrieved to get 20 records is only 17.3.

Table 7

Approximate Retrieval Times with Data Files on 2321 Data Cell

	NUM KEYS	LIST LENGTH	S-LIST LENGTH	QUERY RESP.	INVRTD INDEX	MULTI-LIST	CELL. SERIAL
1.	5	250	50	0	2.6	28.3	.4
2.	5	250	50	1	3.1	28.3	4.5
3.	5	250	50	5	5.4	28.3	21.0
4.	5	250	50	25	16.6	28.3	103.8
5.	5	250	50	45	27.8	28.3	186.5
6.	5	250	200	25	16.6	112.5	103.8
7.	20	1000	200	25	17.5	112.5	103.8
8.	20	1000	200	100	59.6	112.5	414.0
9.	20	1000	800	25	17.5	449.3	103.8
10.	20	1000	800	100	59.6	449.3	414.0
11.	80	4000	800	100	63.2	449.3	414.1
12.	80	4000	800	600	343.9	449.3	2482.3
13.	80	4000	3500	600	343.9	1965.0	2482.3

All Times are in Seconds.

The results of the test cases also make clear several things that were buried within the timing formulas. First of all, the Multilist retrieval times are almost solely a function of the length of the shortest list, and the Cellular Serial retrieval times are almost solely a function of the number of responses to the query.

The Inverted Index retrieval times are influenced by both the number of keys per record and by the number of responses to the query; however, the latter effect appears to be the dominant one.

Note that in every case except the one in which the number of query responses was a high percentage of the shortest list length (Q_r/L_s = 45/50 = 90%), the Inverted Index system proved to be far more efficient than the Multilist system. Similarly

the Inverted Index system dominated the Cellular Serial system in all cases except when the number of query responses was very low.

Note also that in all of the cases the Inverted Index organization could print out the total number of records that satisfy the request within 2.5 seconds

$$(= Q_t T_d + Q_t(P + 1.5R[L/A]))$$

The question arises as to whether the Cellular Serial organization received a fair test in the above examples. Was not Cellular Serial penalized by the slow serial data transmission rate of the 2321 Data Cell, that made it take 4.136 seconds to read an entire cell (magnetic strip), while only .561 seconds to access one record? Table 8 presents the same set of test cases as in Table 7, except that it assumes that all files related to the data base are stored on the 3330 Disk (whose serial data transmission rate is 806 KB/sec vs. 55 KB/sec for the 2321).

The results in Table 8 show that replacing the 2321 Data Cells with the more expensive 3330 Disks cuts the average retrieval times for all three organizations by a factor of about fifteen, with the Inverted Index system still coming out best. This is as expected since both the serial data transmission rate (806 KB/sec for the 3330 vs. 55 KB/sec for the 2321) and also the record accessing time (38ms vs. 561ms) are 15 times better in the 3330. However, all of the organizations become more feasible: using the Data Cell in Case#13 with 80 keys

per record and 600 responses to the query, Cellular Serial would have taken 41 minutes to respond; using the 3330 Disk, Cellular Serial could answer the query in 3 minutes.

Table 8

Approximate Retrieval Times with all Files on 3330 Disk

	NUM KEYS	LIST LENGTH	S-LIST LENGTH	QUERY RESP.	INVRTD INDEX	MULTI-LIST	CELL. SERIAL
1.	5	250	50	0	.5	2.1	.4
2.	5	250	50	1	.5	2.1	.7
3.	5	250	50	5	.7	2.1	1.9
4.	5	250	50	25	1.4	2.1	8.1
5.	5	250	50	45	2.2	2.1	14.3
6.	5	250	200	25	1.4	7.8	8.1
7.	20	1000	200	25	1.4	7.8	8.1
8.	20	1000	200	100	4.3	7.8	31.4
9.	20	1000	800	25	1.4	30.5	8.1
10.	20	1000	800	100	4.3	30.5	31.4
11.	80	4000	800	100	4.5	30.5	31.5
12.	80	4000	800	600	23.4	30.5	186.4
13.	80	4000	3500	600	23.4	132.6	186.4

The results thus far have been for fairly complicated queries involving four keys. What about simpler queries? The simplest types of queries — and in some situations these predominate — involve requests for only a single key (e.g., all employees who are programmers). Note that for a typical single key request $Q_r = L_s = L$. Thus, our thirteen cases presented earlier collapse into just three cases. The results, assuming first the use of the 2321 Data Cell and secondly just the 3330 Disk, are presented in Table 9.

In the case of simple queries there is little to choose from between Inverted Index and Multilist — both give almost

equivalent response times, irrespective of the speed of the direct access storage device. In fact, Multilist does better than Inverted Index, since when the 2321 is used, for every 250 keys in the list, Inverted Index has to retrieve another track of inverted lists at a cost of .075 seconds per track.

Table 9

Approximate Retrieval Times for Simple Queries (Q_t = 1)

A. With Data Files and Inverted Index Lists on 2321 Data Cell

	NUM KEYS	LIST LENGTH	S-LIST LENGTH	QUERY RESP.	INVRTD INDEX	MULTI-LIST	CELL. SERIAL
1.	5	250	250	250	141.0	140.4	1034.2
2.	20	1000	1000	1000	562.2	561.4	4136.5
3.	80	4000	4000	4000	2247.2	2245.5	16545.7

B. With all Files on the 3330 Disk

	NUM KEYS	LIST LENGTH	S-LIST LENGTH	QUERY RESP.	INVRTD INDEX	MULTI-LIST	CELL. SERIAL
1.	5	250	250	250	9.6	9.5	77.6
2.	20	1000	1000	1000	38.0	37.9	310.0
3.	80	4000	4000	4000	151.5	151.4	1239.7

When the average list length is 4000 keys, Inverted Index will take 500 + (4000/250)(.075) = 1.7 seconds more than Multilist. Even this small difference is virtually wiped out when the 3330 Disk is used, since it can hold 1600 addresses per track and since its time to access another track is only .038 seconds. The surprisingly small overhead of the Inverted Index organization can be attributed to our (hopefully realistic) assumption that the entire inverted list for a given key is stored on one

disk cylinder, and thus only one head movement is required.

On the basis of the above results, we now state the following inductively-reasoned hypotheses:

(1) When the queries are complex, the Inverted Index File organization performs retrieval operations far more efficiently than either Multilist or Cellular Serial.

(2) When the queries are simple, the Inverted Index and the Multilist file organizations both perform retrieval operations equally well, and both are more efficient than Cellular Serial.

Several caveats are in order at this point. Contrary to what was just done above, the reader is generally cautioned from making categoric generalizations based on a limited set of results, since such results are often tied to a specific set of "fixed parameter" values. Secondly, retrieval speed is only one factor in selecting a file organization; other important factors that need to be considered are discussed in the next chapter.

Thirdly, the effects of parallel record accessions were not included in the analysis. In the case of a large data base such as in the given example, where ten to twelve IBM 3330 Disk Packs would be needed just to hold the data records, the capability of overlapping disk accessions in satisfying a single user request could imply a substantial decrease in his overall waiting time. In the case of multiple simultaneous users it could imply that their total waiting time would be far less

than the sum of the individual processing times.

Finally, the above results explicitly assumed that records were scattered randomly throughout the data base; an effective method of clustering records according to the key combinations appearing in the most frequently asked queries could make a file organization such as Cellular Serial (where all desired records could be placed contiguously in one or two cells) very attractive. The problem of record clustering will be taken up in more detail in Chapter VII.

CHAPTER V

THE OVERALL FILE ORGANIZATION SELECTION MODEL

5.1 On-line File Updating

In general, the decision to allow on-line updating of a data base should be based on a requirement that update transactions be posted to the files within a short time after they become known. Quantitative cost comparisons between on-line and batch updating are often misleading because of the difficulties in apportioning the fixed costs of storage devices and interactive terminals (which are both needed in any case), and the problem of measuring the hidden costs of decreased response time for on-line users and worsened turnaround for batch users.

On-line updating, once it has been decided upon, can be classified into five categories:

(1) Whole Record Addition
(2) Whole Record Deletion
(3) Deletion of Keys
(4) Addition/Deletion/Modification of Non-key Data
(5) Addition of Keys

The designer of a file organization that allows update must give attention to two problems. The first is the updating of the inverted lists or the internal list pointers

whenever a key is involved. This may occur in all of the above categories except (4). The second is the relocation of a record when an update expands its size, and the subsequent utilization of its former space. This may occur in Categories (4) and (5). The procedures for effecting these updates and handling these two major problems are somewhat different for threaded and inverted lists.

David Lefkovitz in his book File Structures for On-line Systems (49) has an excellent chapter in which he describes the updating procedures for the Multilist and Inverted Index file systems. His descriptions are extremely clear, and they have the advantage of being written in a consistent style. For this reason, only the principal actions that need be taken for each category of file updating are summarized here. For a more detailed description, the reader is encouraged to go directly to Chapter VIII of Lefkovitz. However, Lefkovitz's discussion of the on-line updating of the Cellular Serial system is quite limited, and a more comprehensive description of the updating techniques for that system is presented here.

5.2 Updating Multilist Files

At the beginning of the record is a single bit called the Record Delete Bit. If this bit is set to 0 it means that the record is in the file; if it is set to 1, the record has been logically, though not physically, deleted from the file. In addition, associated with each Key/Link Address field is a Key Delete Bit (similarly used). The Key/Link Address pair

cannot actually be removed from the record because the list linkage would then be broken, and the alternative of transferring the pointer to the link address of the previous record would require a bi-directional list, which is very costly in terms of space consumption.

(1) Multilist Files — Whole Record Deletion

Access the record and set the Record Delete Bit to 1. Since it is preferable that the key list lengths within the directory be viewed as being the physical length of the lists, (i.e., the actual number of random accessions needed to traverse the list), nothing further need be done. Maintaining a physical interpretation of the list lengths is useful for determining which list in a query product should be searched.

(2) Multilist Files — Deletion of Individual Keys Within a Record

Set the Key Delete Bit to 1 for the key(s) to be deleted. The actual purging of deleted keys and deleted records and the reclaiming of the space is performed during off-line file maintenance.

(3) Multilist Files — Whole Record Addition

Assign a storage device address AD to the new record. For each key in the record, transfer the current head-of-list address from the directory to the link address of that key

in the new record, make AD the new head-of-list address for that key, and increment the list length of that key in the directory by one.

(4) Multilist Files — Addition/Deletion/ Modification of Non-key Data

Read the record from disk into core and modify the record. If the record is shortened, repack the track and write it onto the disk. If the record is increased in size and the repacked track does not overflow, then write it back onto the disk. If the record is increased and the repacked track overflows, then delete the whole record and insert the modified record onto the disk according to the procedure for Whole Record Addition.

(5) Multilist Files — Addition of Individual Keys to a Record

Read the record from disk into core and add new key(s). Follow the same procedure for adding each key as is used in Whole Record Addition. Restore the modified record according to the procedure used for Modifying Non-key Data.

5.3 Updating Inverted Index Files

(1) Inverted Index Files — Whole Record Deletion

Access the record and set the Record Delete Bit to 1. Decode every key of the record and remove the record address from every inverted list on which it appears, repacking the

shortened key lists and restoring them to disk. Decrement the key list lengths by one.

(2) Inverted Index Files — Deletion of Individual Keys Within a Record

Decode each key to be deleted from the record and remove the record address from the key list. Repack the shortened key lists, restore them to disk, and decrement the list lengths by one. Note that Key Delete Bits are not used with Inverted Index because the key links do not exist on the file area itself, and hence deletion of a key from a record does not destroy the continuity of the list.

(3) Inverted Index Files — Whole Record Addition

Assign a storage device address AD to the new record. Decode each key in the directory to its proper inverted list and insert the address AD in sequence into the list. If the insertion of this address causes the list to overflow the allocated block on mass storage, attach another block and chain it to the previous one. Update the first record of the inverted list by incrementing the list length by one. This is done for each key in the record.

(4) Inverted Index Files — Addition/Deletion/ Modification of Non-key Data

The updating of non-key data is basically identical to that in the Multilist system since no key lists are involved. The only

exception to this occurs when the record is increased in size and the repacked track overflows, then the process of deleting the old record and inserting the modified record onto a new track (using Whole Record Addition) involves the updating of all the lists associated with the record.

(5) Inverted Index Files — Addition of Individual Keys to a Record

Read the record from disk into core and add new key(s). Follow the same procedure for adding each key as is used in Whole Record Addition. Restore the modified record according to the procedure used for Modifying Non-key Data.

5.4 Updating Cellular Serial Files

In the Cellular Serial System the inverted list for a given key consists of a list of those cells that have one or more records containing that key. The process of adding a new key to a record in the cell requires that the inverted list be examined to see whether the given key is already contained in the cell. If so, the inverted lists need not be changed; if not, the cell that will be getting the new key must be added to the inverted list for that key.

(1) Cellular Serial Files — Whole Record Deletion

Access the record and set the Record Delete Bit to 1. However, this is all that is usually done on-line. The updating of the inverted lists is performed during file maintenance, since the

entire cell must be read and the keys of all the records in the cell analyzed to determine whether the cell address should be dropped from the list of any key. If the only occurrence of a given key value in the cell was in the deleted record, then the cell is removed from the inverted list.

(2) Cellular Serial Files — Deletion of Individual Keys Within a Record

The process of key deletion in Cellular Serial systems is similar to that of whole record deletion; only the setting of the Key Delete Bit is performed on-line. The reading of the entire cell to determine whether the specified key has been retired from the cell is again performed off-line during file maintenance.

(3) Cellular Serial Files — Whole Record Addition

The process of Whole Record Addition requires that the inverted list of each key of the new record be examined to see whether the parent cell of the new record already has that key. In many cases only a small fraction of the keys in a new record will have to have their inverted lists modified to include a new cell. In general, the smaller the ratio of the number of keys per cell to the total number of unique keys in the system, the greater the probability that some of the key lists will have to be updated.

(4) Cellular Serial Files — Addition/Deletion/ Modification of Non-key Data

Since it is such a time-consuming process when the first key of a

given type is added to a cell, an effort is made to keep record relocations caused by cell overflow to a minimum. This is usually accomplished by leaving sufficient reserve space at the end of the cell, or distributed throughout the cell, that the great majority of record relocations due to expansion are kept within the same cell. For this reason, the Addition/Deletion/Modification of Non-key Data is usually a straightforward process. If cell overflow does occur, the old record is deleted according to the Whole Record Deletion process, and then inserted in another cell by way of Whole Record Addition.

(5) Cellular Serial Files — Addition of Individual Keys Within a Record

Access the record and add the new key(s). Follow the same procedure for adding each key as is used in Whole Record Addition. As was the case with Modifying Non-key Data, any expansion of the record can usually be accommodated within the same cell.

5.5 File Update Timing Formulas

The development of accurate update timing formulas is a difficult problem — particularly so when it is done in the abstract without a real data base to fall back on for substantiation. Siler (83) in his timing simulations specifically avoided simulating any updating procedures. Cardenas (13) also avoided updating factors, specifically mentioning that significant analytic work is needed in order to properly account for such factors. He stated that the update formulas derived by others

(specifically Lefkovitz and Martin) were oversimplified in that they did not take into account the various possible side effects which depend on the characteristics of the data management routines of the specific operating system (e.g., the overflow effects of the indexed sequential methods).

Such problems aside, an attempt is made here to develop various update timing formulas under the assumption that approximate or even imperfect formulas are better than no formulas at all. The rationale behind many of the timing formulas will usually be evident after a close reading of Sections 5.1-5.4. Most of the update timing formulas presented here are based on the work of Lefkovitz; however, they differ in at least six key areas from those proposed by Lefkovitz:

(1) Lefkovitz assumed that the time needed to update the directory was equal to the time to decode the directory. This is generally not true. When adding or deleting an address to a key list in the Inverted Index system, or when adding a record to a key list in the Multilist system, the lowest level of the directory is updated so that the list length for that key can be incremented or decremented by one. Thus, the time to update the directory $T_u = T_d + 1.5R$. In the rare case when an entirely new key is added to the system, every level of the directory must be updated, thus $T_u = 2T_d$. For the Cellular Serial system, the directory does not keep track of how many records containing a key are in each cell, so it does not need to be updated except when a brand new key is added to the system.

(2) When the Multilist and Inverted Index systems modify non-key data such that record relocation is required, Lefkovitz allowed only one disk access to store the updated record. Two accesses appear to be needed: one to purge the old version of the record from its current track, and one to store the updated record on its new track.

(3) For whole record or single key addition in the Cellular Serial system, Lefkovitz seems to have ignored the fact that the directory and the inverted lists need to be searched, and that occasionally (the first time a given key is brought into the cell) the inverted lists need to be updated. The timing formulas presented here allow for such processing.

(4) When adding or deleting a record address to an inverted list, the list must be read up to the point at which the address should be located, the new address must be inserted, and the one track of the revised list must be rewritten. This requires that an average of one half of the inverted list be read for each key. Since he allowed for only one head movement, Lefkovitz implicitly assumed that the entire inverted list for a given key was located on the same cylinder. However, this contradicts the assumption he made in his retrieval timing analysis, where he allowed a full head positioning movement for each track that contained part of the Inverted List. The update and the retrieval timing formulas presented in this paper are consistent in that both assume that all the inverted list tracks for a given key are located within the same cylinder.

(5) When modifying non-key data in the Multilist system and the record expands enough to cause the track to overflow, the update is handled by setting the Record Delete Bit in the old record and by inserting the expanded record into a new track. This requires the lowest level of the directory to be updated with the list length incremented by one, since purged records physically remain in the list and must be included in the count to ascertain the shortest list in conjunctions.

(6) The modifying of non-key data in the Inverted Index system such that record relocation is required is a tricky case. The list length in the directory need not be updated since one record is being purged while an expanded version of the record is being added. The directory and the inverted lists need be read only once for each key since the old address can be purged and the new address of the record can be inserted in one pass. But how many tracks of the inverted list must be read to find one key address and delete it, and to add another in sequence? To simplify matters, it will be assumed that the address of the new record is located on the same track as the address of the old record. In the case of the 3330 Disk, which can have 1600 link addresses on one track, this is a reasonable assumption any time the list length is less than 3000.

The following notation is used in conjunction with the update timing analysis. It is basically the same as that

used for the retrieval timing analysis:

N_k = Number of keys/record (average)

T_d = Time required to decode the directory

T_u = Time required to update the directory = T_d+1.5R

T_a = Time required to access a record = P+.5R+C_f/R_t

T_i = Time required to read halfway through Inverted Index's inverted list and update the appropriate track

= P + 1.5R + 1.5R if [L/A] = 1

= P + .5 [L/A] 1.5R + 1.5R if [L/A] > 1

T_c = Time required to read halfway through Cellular Serial's inverted list and update the appropriate track

= P + 1.5R + 1.5R(f) if $[C_k/A]$ = 1

= P + .5 $[C_k/A]$ 1.5R + 1.5R(f) if $[C_k/A]$ > 1

f = The percentage of time that a key is to be added to a cell in which it is not already represented (only relevant for Cellular Serial)

Table 10 breaks down the timing for each type of update into its basic steps; Table 11 summarizes the overall update timing formulas for each type of organization. These tables are constructed assuming that all updates (e.g., addition of n keys) are made against a single record. In the case of generic updates, appropriate multipliers are required to account for multiple key decoding, record accession, directory updating, and record updating. For example, a generic

TABLE 10: UPDATE TIMING

	PROCESS	WHOLE RECORD ADDITION	WHOLE RECORD DELETION	DELETION OF n KEYS	NON-KEY MODIFIC. CASE A	NON-KEY MODIFIC. CASE B	ADDITION OF n KEYS CASE A
Multilist Organization	Decode Directory		T_d	T_d	T_d	T_d	T_d
	Access Record		T_a	T_a	T_a	T_a	T_a
	Update Directory	N_kT_u				N_kT_u	nT_u
	Store Updated Record	T_a	T_a	T_a	T_a	$2T_a$	T_a
Inverted Index Organization	Decode Directory		T_d	T_d	T_d	T_d	T_d
	Access Record		T_a	T_a	T_a	T_a	T_a
	Update Directory	N_kT_u	N_kT_u	nT_u		N_kT_d	nT_u
	Update Inverted List	N_kT_i	N_kT_i	nT_i		N_kT_i	nT_i
	Store Updated Record	T_a	T_a	T_a	T_a	$2T_a$	T_a
Cellular Serial Organization	Decode Directory		T_d	T_d	T_d	T_d	T_d
	Access Record		T_a	T_a	T_a	T_a	T_a
	Update Directory	N_kT_d					nT_d
	Update Inverted List	N_kT_c					nT_c
	Store Updated Record	T_a	T_a	T_a	T_a	T_a	T_a

CASE A = WITHOUT RECORD RELOCATION CASE B = WITH RELOCATION

TABLE 11

UPDATE TIMING COMPARISONS AMONG THE THREE FILE STRUCTURES

UPDATE TYPE	MULTILIST	INVERTED INDEX	CELLULAR SERIAL
Whole Record Addition	$T_a+N_kT_u$	$T_a+N_k(T_u+T_i)$	$T_a+N_k(T_d+T_c)$
Whole Record Deletion	T_d+2T_a	$T_d+2T_a+N_k(T_u+T_i)$	T_d+2T_a
Deletion of n Keys	T_d+2T_a	$T_d+2T_a+n(T_u+T_i)$	T_d+2T_a
Non-Key Modification (w/o Relocation)	T_d+2T_a	T_d+2T_a	T_d+2T_a
Non-Key Modification (with Relocation)	$T_d+3T_a+N_kT_u$	$T_d+3T_a+N_k(T_d+T_i)$	T_d+2T_a*
Addition of n Keys	$T_d+2T_a+nT_u$	$T_d+2T_a+n(T_u+T_i)$	$T_d+2T_a+n(T_d+T_c)$

*It is assumed that sufficient space is left within the cell such that relocation is very rare.

T_d = Time required to decode the Directory

$T_u = T_d + 1.5R$

$T_a = P + .5R + C_f/R_t$

$T_i = P + .5[L/A]1.5R + 1.5R$

$T_c = P + .5[C_k/A]1.5R + 1.5R(f)$

update might be something like: "For all employees with TITLE = CHEMIST, change SUPERVISOR to SMYTHE J."

5.6 Examples Using the Updating Formulas

The FORTRAN program used earlier in this report for calculating retrieval times has also been designed to calculate the various update times for the same test cases. However, note from the update formulas that of the three critical parameters isolated for retrieval time — N_k, L_s, and Q_r — only N_k (the average number of keys per record) has an effect on updating time. In fact, once the characteristics of the data base and the specifications of the DASD have been set, only N_k and f (the fraction of keys to be added that are not already contained in the cell) have an effect on update times, and f is relevant only for the Cellular Serial organization.

Tables 12 and 13 show the output of the program for the parameter values specified in Table 6, with f set at the reasonable value of .2 and N_k set at the three different values used earlier for retrieval timing (5, 20, and 80). Table 12 is based on the same assumptions as Table 8: all directories and the inverted indexes of the Cellular Serial organization are assumed to reside on the 3330 disk, while all data base files and the inverted lists of the Inverted Index system are assumed to be on the 2321 data cell. Table 13 is based on the same assumptions as Table 9: all files are assumed to be located on a 3330 disk.

Table 12: Update Times with Data Files on 2321 Data Cell

NUM KEYS	LIST LENGTH	INVRTD INDEX	MULTI-LIST	CELL. SERIAL	UPDATE OPERATION
5	250	4.3	1.0	1.2	REC.ADDITION
		4.9	1.2	1.2	REC.DELETION
		1.2	1.2	1.2	MOD W/O RELOC.
		5.3	2.2	1.2	MOD WITH RELOC.
		2.7	1.2	1.2	DELETE 2 KEYS
		2.7	1.4	1.4	ADD 2 KEYS
20	1000	16.9	2.4	3.1	REC.ADDITION
		17.6	1.2	1.2	REC.DELETION
		1.2	1.2	1.2	MOD W/O RELOC.
		17.6	3.6	1.2	MOD WITH RELOC.
		2.8	1.2	1.2	DELETE 2 KEYS
		2.8	1.4	1.4	ADD 2 KEYS
80	4000	102.1	8.1	11.6	REC.ADDITION
		102.7	1.2	1.2	REC.DELETION
		1.2	1.2	1.2	MOD W/O RELOC.
		101.3	9.3	1.2	MOD WITH RELOC.
		3.7	1.2	1.2	DELETE 2 KEYS
		3.7	1.4	1.5	ADD 2 KEYS

It is obvious from Table 12 that whereas Inverted Index is by far the most efficient organization for record retrieval, it is by far the least efficient organization for on-line updating. The principal reason why Inverted Index performed so poorly was that its long inverted lists were located on the slow 2321 Data Cell. In the 80-key case, of the 102.1 seconds needed to perform a record addition 94 seconds was consumed in updating the inverted lists alone (excluding the directory updating): 40 seconds (80 keys x 500 ms/key) was needed to position the mechanism at the start of the inverted lists; 48 seconds (80 keys x ½ x 16 tracks x 75 ms/track) was needed to read halfway through each key list, which averaged 4000/250 = 16 tracks in length; and 6 seconds (80 keys x 75 ms/key) was

needed to update the appropriate track on each key list.

The assumptions used in generating Table 12 can be modified in a couple of different ways to provide additional insights. For example, Table 12 was based on the optimistic assumption that the entire inverted list for a given key was placed on the same cylinder (or at least on the same magnetic strip). In a highly volatile environment with numerous key changes and heavy record additions this might prove to be almost impossible to maintain. Moreover, in a situation where no concern whatsoever was given to initially placing the inverted lists on the same cylinder, the time to add one 80-key record would be over 370 seconds.

On the other hand, if the eventual users were willing to pay more money for mass storage, the inverted lists could be put on the IBM 3330 disk along with the directories, leaving only the data base records themselves on the data cell. In this situation it would take only 7.0 seconds to update all the inverted lists of an 80-key record (versus 94.0 seconds before), and only 15.1 seconds overall to add an entire 80-key record. Part of this dramatic decrease in time is accomplished by the faster positioning and rotation speeds of the 3330, and part by the fact that it can hold more record addresses per track than the 2321.

The results shown in Table 13 were generated assuming that all files connected with the data base are placed on 3330 disks. Note that although all of the file organizations perform

updating more efficiently when only the 3330 disks are used, it is the Inverted Index organization that exhibits the most drastic improvement. As discussed previously, much of this gain is due to moving the inverted lists to the disk; for example, in deleting an entire record, 87 seconds are gained by transferring the inverted lists to disk and slightly less than one second is gained by transferring the data records to disk too.

Table 13: Update Times with all Files on 3330 Disk

KEYS	LENGTH	INVRTD INDEX	MULTI-LIST	CELL. SERIAL	UPDATE OPERATION
5	250	.9	.5	.7	REC.ADDITION
		1.0	.1	.1	REC.DELETION
		.1	.1	.1	MOD W/O RELOC.
		.9	.7	.1	MOD WITH RELOC.
		.5	.1	.1	DELETE 2 KEYS
		.5	.3	.4	ADD 2 KEYS
20	1000	3.5	1.9	2.6	REC.ADDITION
		3.6	.1	.1	REC.DELETION
		.1	.1	.1	MOD W/O RELOC.
		3.1	2.1	.1	MOD WITH RELOC.
		.5	.1	.1	DELETE 2 KEYS
		.5	.3	.4	ADD 2 KEYS
80	4000	14.7	7.5	11.1	REC.ADDITION
		14.8	.1	.1	REC.DELETION
		.1	.1	.1	MOD W/O RELOC.
		12.9	7.7	.1	MOD WITH RELOC.
		.5	.1	.1	DELETE 2 KEYS
		.5	.3	.4	ADD 2 KEYS

The updating times presented here cannot be directly compared with the retrieval times shown earlier, because the latter depend on factors, such as the number of terms in the

query and the number of responses to the query, that have no direct parallel in file updating. However, a rough comparison shows that except for the Inverted Index system the retrieval operations generally take substantially longer than the updating operations. This is true whether comparing Tables 7 and 12 or Tables 8 and 13 (for parallel assumptions).

Again the reader is cautioned against making generalizations without going back to the basic timing formulas. For example, Tables 12 and 13 might lead one to believe that adding or deleting keys to a record is inherently a more efficient operation than the addition or deletion of an entire record. However, this appears to be true only because the chosen example illustrates the time needed to add 2 keys to a given record. It turns out that for all three file organizations the time needed to add 20 keys to an existing record is greater than the time needed to add an entire 20-key record.

5.7 Selecting the "Best" File Organization

One of the main purposes of this paper is to formulate some general decision rules for choosing the optimum file organization for any given case. It would appear that after the preceding detailed analysis of retrieval and update times, we are well prepared to do just that. Unfortunately, the answer to the "what is best" question still appears to be very complex and qualitative. As mentioned earlier, Cardenas (13) suggested in a recent paper that the best data base system is one that minimizes a cost function that includes —

(a) On-line retrieval costs

(b) On-line updating costs

(c) Storage costs

(d) Off-line file maintenance and file updating costs

(e) Costs for loading the data base initially and for performing file re-organization at appropriate intervals to improve the performance of the system

(f) Other highly unquantifiable costs including the costs of initial programming, re-programming, documentation, data base administration, etc.

— while satisfying given design constraints (such as the query response time must be less than 3 seconds for 80% of the queries).

We can make some general observations as to what will happen to each of the individual cost functions in future years. The on-line operating costs include not only the computer charges, telephone line costs, and interactive terminal costs, but also the costs of having people sitting on-line waiting for answers. People costs have risen sharply over the past few years and will continue to do so. With data bases growing larger, more all-encompassing, and capable of handling more simultaneous users, more and more people will be sitting on-line and waiting.

Computer costs, on the other hand, have fallen sharply over time, while computer capacities have risen dramatically; these trends should both continue. Thus, the fixed costs of the direct

access storage devices, of the interactive and remote batch terminals, and of the computer time needed to load the data base initially and to perform periodic file reorganizations and regular batch-mode file maintenance and updating should be declining in the future.

The software development and maintenance costs for major data bases have historically been both large and grossly underestimated. However, two trends are operating to reduce that problem. First, with so much now known and documented about designing data bases, most in-house development projects in this area will no longer be pioneering. The success of some very inexpensive data bases — O'Connell (66) — is testimony to the validity of this argument. Secondly, the generalized data base management systems are getting more sophisticated, more user-oriented, and more prolific all the time; and relative to the general rate of inflation, the cost of these generalized systems is declining.

With software development costs and computer-related costs going down relative to the costs of keeping people waiting, we propose as a useful first cut decision rule the minimization of the average on-line processing time per operation. This is defined as follows:

$$\text{Average on-line processing time} = P_1(\text{RET}) + \sum_{i=2}^{n} P_i(\text{UPDAT}_i)$$

where

$$\sum_{i=1}^{n} P_i = 1$$

RET = estimated average retrieval time

$UPDAT_i$ = estimated average processing time for update operation type i

P_i = percentage of all on-line operations that consists of type i processing

Martin (57) developed a similar, although simpler, model that was based on less exact timing formulas. However, as mentioned in Chapter III, he contorted the resulting equations in such a way as to derive an ultimate model that was less useful than the original one.

Examples are presented in Tables 14 and 15 illustrating the use of the above decision rules. To simplify the examples only three of the several possible updating operations have been included:

P_2 = percentage of whole record additions

P_3 = percentage of whole record deletions

P_4 = percentage of non-key data modifications (without record relocation)

Table 14: Some Results Using the Decision Rules with the Data Files on the 2321 Data Cell

CASE 1 N_k = 5 L_s = 50 Q_r = 5

P1	P2	P3	P4	INVRTD INDEX RETRVL	INVRTD INDEX OVRALL	MULTILIST RETRVL	MULTILIST OVRALL	CELL. SERIAL RETRVL	CELL. SERIAL OVRALL
.90	.03	.03	.04	5.4	5.2	28.3	25.6	21.0	19.0
.60	.05	.05	.30	5.4	4.0	28.3	17.4	21.0	12.7
.50	.20	.20	.10	5.4	4.6	28.3	14.7	21.0	11.0
.10	.10	.10	.70	5.4	2.3	28.3	3.9	21.0	2.3
.10	.50	.30	.10	5.4	4.3	28.3	3.8	21.0	3.1

CASE 2 N_k = 20 L_s = 200 Q_r = 25

P1	P2	P3	P4	INVRTD INDEX RETRVL	INVRTD INDEX OVRALL	MULTILIST RETRVL	MULTILIST OVRALL	CELL. SERIAL RETRVL	CELL. SERIAL OVRALL
.90	.03	.03	.04	17.5	16.8	112.5	101.4	103.8	93.5
.60	.05	.05	.30	17.5	12.6	112.5	68.0	103.8	62.5
.50	.20	.20	.10	17.5	15.8	112.5	57.1	103.8	52.7
.10	.10	.10	.70	17.5	6.0	112.5	12.4	103.8	10.8
.10	.50	.30	.10	17.5	15.6	112.5	12.9	103.8	12.3

CASE 3 N_k = 80 L_s = 3500 Q_r = 600

P1	P2	P3	P4	INVRTD INDEX RETRVL	INVRTD INDEX OVRALL	MULTILIST RETRVL	MULTILIST OVRALL	CELL. SERIAL RETRVL	CELL. SERIAL OVRALL
.90	.03	.03	.04	343.9	315.7	1965.0	1768.8	2482.3	2234.5
.60	.05	.05	.30	343.9	216.9	1965.0	1179.8	2482.3	1490.0
.50	.20	.20	.10	343.9	213.0	1965.0	984.5	2482.3	1243.7
.10	.10	.10	.70	343.9	55.7	1965.0	198.3	2482.3	249.5
.10	.50	.30	.10	343.9	116.3	1965.0	201.0	2482.3	254.4

The FORTRAN program used earlier for calculating the retrieval and updating examples includes a section illustrating the use of the overall model for five very different situations. For example, in the first situation 90% of the on-line operations are retrievals, with the other 10% being divided between additions, deletions, and modifications; in

the fifth, only 10% of the operations are retrievals, while 50% are record additions and 30% are record deletions. Table 14 is based on the results shown in Tables 7 and 12; it assumes the use of IBM 3330 disks and IBM 2321 data cells as earlier described. Table 15 is based on the results shown in Tables 8 and 13; it assumes that only the 3330 disks are used. Both tables use three of the more representative cases selected from the thirteen presented in Tables 7 and 8.

Table 15: Some Results Using the Decision Rule with All Files on the 3330 Disks

CASE 1 N_k = 5 L_s = 50 Q_r = 5

				INVRTD INDEX		MULTILIST		CELL.	SERIAL
P1	P2	P3	P4	RETRVL	OVRALL	RETRVL	OVRALL	RETRVL	OVRALL
.90	.03	.03	.04	.7	.7	2.1	1.9	1.9	1.7
.60	.05	.05	.30	.7	.5	2.1	1.3	1.9	1.2
.50	.20	.20	.10	.7	.7	2.1	1.2	1.9	1.1
.10	.10	.10	.70	.7	.4	2.1	.4	1.9	.3
.10	.50	.30	.10	.7	.8	2.1	.5	1.9	.6

CASE 2 N_k = 20 L_s = 200 Q_r = 25

				INVRTD INDEX		MULTILIST		CELL.	SERIAL
P1	P2	P3	P4	RETRVL	OVRALL	RETRVL	OVRALL	RETRVL	OVRALL
.90	.03	.03	.04	1.4	1.5	7.8	7.1	8.1	7.4
.60	.05	.05	.30	1.4	1.3	7.8	4.8	8.1	5.0
.50	.20	.20	.10	1.4	2.1	7.8	4.3	8.1	4.6
.10	.10	.10	.70	1.4	.9	7.8	1.1	8.1	1.1
.10	.50	.30	.10	1.4	3.0	7.8	1.8	8.1	2.1

CASE 3 N_k = 80 L_s = 3500 Q_r = 600

				INVRTD INDEX		MULTILIST		CELL.	SERIAL
P1	P2	P3	P4	RETRVL	OVRALL	RETRVL	OVRALL	RETRVL	OVRALL
.90	.03	.03	.04	23.4	21.9	132.6	119.6	186.4	168.1
.60	.05	.05	.30	23.4	15.5	132.6	80.0	186.4	112.4
.50	.20	.20	.10	23.4	17.6	132.6	67.9	186.4	95.5
.10	.10	.10	.70	23.4	5.4	132.6	14.1	186.4	19.8
.10	.50	.30	.10	23.4	14.2	132.6	17.1	186.4	24.2

Tables 14 and 15 demonstrate that the retrieval efficiency of the Inverted Index file is significant enough in most cases to keep its average on-line processing time well below that of the other file organization strategies. The Inverted Index system was by far the most efficient when retrieval operations made up 90%, or even 50% of the on-line operations. In the cases where non-key modifications predominated, Inverted Index still proved to be best, although Cellular Serial performed as well when the average list length (which is directly related to the number of keys/record in these cases) was low. Only in the case where the relative percentage of retrieval operations (10%) was overshadowed by the percentage of record additions (50%) and deletions (30%) did the other systems prove to be more efficient, and then only where the average list lengths were low or medium.

Once again the reader is cautioned against overgeneralizing from the above results. The results are indicative, rather than definitive. Although a number of realistic test cases were examined, it must be remembered that all of them utilized the same basic set of "fixed" parameter values (e.g., number of records, number of terms/query, transfer rate of the DASD, etc.). A new situation in which any of the parameter values are different should be run through the decision rules. For example, it should be noted that in all

cases presented in Tables 14 and 15 the percentage of record relocations caused by adding keys or non-key data was assumed to be zero. Such record relocations are far more time-consuming to perform under Inverted Index than under either of the other two organizations.

CHAPTER VI

DETAILED IMPLEMENTATION TECHNIQUES AND TRADE-OFFS

The first five chapters of this thesis were designed to emphasize the macroscopic problem of selecting the best file organization in a given situation. The purpose of this chapter is to get closer to the "nitty-gritty" problems of actually implementing a multi-key retrieval system. The intent is not to reproduce a minute detail of all the specific techniques that have been proposed in the literature, but rather to organize the techniques according to a useful framework, to discuss the primary trade-offs that exist between alternatives, and to provide the interested reader with "link pointers" by which he can pursue specific techniques in greater detail.

In general, most of the implementation alternatives will be discussed within the framework of the Inverted Index file structure. This is because the Inverted Index system has proven to be both complicated, thus allowing many possible design alternatives, and popular, implying that many of the alternatives have been proposed in the literature. Moreover, the test results described in Chapters IV and V of this thesis strongly indicate that in a wide variety of situations Inverted Index provides both the fastest retrieval times and the fastest average on-line processing times. The first general area to be treated — the organization of the key directory — is equally applicable to any of the organizations

discussed thus far.

6.1 Techniques for Directory Decoding

The purpose of a directory is to translate a key taken from a query to an address that points to an inverted or threaded list of record addresses containing that key. One of the most complete discussions of directory decoding techniques is presented by Lefkovitz (49). He discusses four types of decoders: one based on fixed length key fragments obtained by truncation; one based on full (variable) length keys; one based on truncating keys to unique variable length fragments; and one based on hash coding the keys onto address locations. The first three decoders all utilize a balanced tree to decode the key values.

6.1.1 The Balanced Tree

The balanced tree has the property that if the lowest level of the tree is N, then all keys can be decoded in either (N-1) or N levels, thus giving rise to a nearly constant decoding time. Assuming that the number of key/address pairs that can fit on a track (tree node) is M, then the maximum number of unique keys that can be decoded in an N-level tree is M^N. M is often referred to as the tree branching factor. For example, a 3-level tree having 200 keys per track could decode up to $(200)^3$ = 8 million keys. This means that a unique key value out of 8 million could be decoded in three random accessions, and in two accessions if the top level of the tree were maintained in core. Landauer (48) wrote one of

the earliest papers describing the use of balanced trees in the decoding of keys.

6.1.2 Fixed Length Key Fragments

The most popular type of decoder is the balanced tree using fixed length key fragments obtained by truncating the full length keys. The principal advantages of this technique are its programming simplicity, its decoding speed, and its ability to perform range searches on keys (e.g., AGE/21 THRU AGE/30). Its main disadvantage is the fact that it allows ambiguous decoding of keys. The ambiguity can be resolved either by examining the file records, thus resulting in excessive retrievals, or by including the full key somewhere in the directory, thus leading to a higher storage overhead. SDC's sophisticated Time-Shared Data Management System (TDMS) utilized fixed length truncated keywords, and set an indicator any time there were duplicate key values on either side of a particular key fragment (Bleier and Vorhaus (8)).

6.1.3 Full Variable Length Keys

The second technique involves the use of the full, variable length keys and thus assures completely unambiguous decoding, but at the cost of increased storage consumption. In particular, decoding full keys using a balanced tree is simple to program, but wasteful of disk storage. Decoding full keys with a standard parsing tree (see Sussenguth (86)) is more economical in storage, but more complex to program.

6.1.4 Variable Length Key Fragments

The third technique is based on truncating the keywords to variable length unique fragments. For example, if BABBET, BABSON, and BAILEY are the first three keys, the first would be encoded to the key fragment B, the second to the key fragment BABS (since BABBET and BABSON differ at the fourth letter), and the third to the fragment BAI. This technique is substantially more difficult to program than the fixed length truncated keywords, but it decodes all keys unambiguously. However, its core and disk storage requirements are equal to that of the fixed length truncated keywords when the latter's special storage requirements for resolving ambiguity are taken into account, and substantially greater when the ambiguous decoding of the fixed tree is resolved by extra file accessions.

Wagner (90) has reported on a related, but more complicated, technique involving both front and rear compression. For example, the distinction between BABBET and BABSON would be the B and the S respectively in the fourth position. Wagner has also suggested the use of pointer compression based on relative addressing in the case where tree nodes are fixed-size tracks. Thus, if only 200 nodes can be addressed from a given node, then only 8 bits need be used for addressing (since $2^8 = 256 > 200$).

6.1.5 Hash Coding Techniques

The fourth technique discussed by Lefkovitz involves the use of hash coding (or randomizing) the key words to specific memory locations, with chaining being used to take care of

multiple key hits on the same location. The randomizer decoders generally require less core and disk storage than the various truncation techniques (particularly when the overhead needed to resolve ambiguity is considered) and often decode in less time. However, they can become inefficient when the chains grow long, and they are not capable of automatically performing range searches (e.g., $10,000 \leq \text{SALARY} \leq 25,000$). Section 3.1 of this thesis includes a fairly good bibliography on hash coding techniques, many of which involve open addressing methods within the hash coding area, a technique that should prove more effective in a disk environment than Lefkovitz's proposed chaining techniques. The use of hash coding to locate the inverted lists for a given key has also been described by Bloom (9).

6.2 Techniques for Organizing the Data Base Files

6.2.1 Using Record Identity Numbers

Several authors — notably Lefkovitz (49), Prywes (71), and Bloom (9) — have discussed the interval structuring of the data base records. Most of them agree on the appropriateness of assigning each record a unique "Record Identity Number" (RIN) or record accession number. These RINs may have a physical connotation (e.g., social security numbers, invoice numbers) or may be assigned sequentially as records are added to the data base. Bloom suggests that in certain applications it may be possible to store the records on the disk more or less serially so that there is a simple relation between the

RIN and the disk location of the record. In other cases the data files may be organized as direct access files based on hash coding the RIN, or as indexed sequential files using the RIN as the primary key.

If the RIN is an actual data base key, retrievals and updates involving that key are made more efficient because of the juxtaposition of all records having a given key value. However, whether the RIN is a natural key or not, it is useful to batch off-line file updates by RIN (or by the hash code of the RIN if direct access is used), thus keeping the head positioning movements between data records to a minimum. The disadvantages of the RIN approach are its extra storage requirements when the RIN is not a natural key, and the loss of efficiency in sequentially processing the data base when the optimum key for sequential ordering is not unique.

6.2.2 Intra-record Organization of Data Base Records

Within each variable length data record are the RIN, the individual key names and associated key values, and the non-key data of the record. A table of contents is included within the record to indicate the starting addresses for each key name/key value combination and the starting point of each non-key data component. Often the key names are identified by short codes, which can then be converted back to the full key names for printing by means of a small core resident table. In the various threaded list organizations each key will also have associated with it a pointer to the next record

on its chain. In addition, the record may contain usage statistics relating to the entire record, or even to detailed keys, or may have access control keys that allow the system to restrict query access to a given record via a code presented in the query.

6.2.3 Using a Separate Table for Literal Strings

Apart from these basic specifications for the internal record structure, there are several areas in which the file designer can make a choice between alternatives that have different implications in terms of retrieval and updating efficiency, storage costs, and programming complexity. For example, long alphanumeric strings can either be stored within the record or removed to a separate table and referenced by pointers. For large data bases in which the alphanumeric strings have many duplicates (e.g., CITY = "LOS ANGELES, CALIFORNIA"), considerable storage space can be saved by listing these literals only once in the separate table instead of many times within the data base records. This is the approach used by TDMS (8) and by Bloom (9). The disadvantage of this approach is that the retrieval of the literal data requires additional file accessions beyond the retrieval of the basic data records.

6.2.4 Removing the Keys from the Data Records

Another decision area that was mentioned both by Cardenas (13) and Lefkovitz (49) is the problem of whether or not the keys themselves should be stored in the data records.

If the keys are not needed for query output, then they can be omitted from the data records entirely. Thus, each key will be listed only once in the entire data base (at the level of the directory output that points to the inverted lists). This option can save considerable storage with the Inverted Index system, bringing its disk memory requirement below that of any of the other file organizations discussed. However, if the entire records have to be reconstructed to answer certain queries, then a whole series of links would have to be provided. This would entail some sophisticated programming and also the degradation of the query response time.

6.2.5 Alternatives for Handling Record Overflow

A third decision area is that of handling the problem of a record that expands and causes the track to overflow. Lefkovitz (49) handles this problem by deleting the original record and placing the expanded version on a new track. This necessitates changing the record address of that record in the inverted lists of all keys contained in the record. The space occupied by the deleted record is reclaimed during off-line file maintenance. Another alternative when a record expands too much is simply to add a trailer record on another track. This eliminates the need to modify the key lists, but requires that each future retrieval of that record access two or more tracks.

Assuming that the trailer record can be located on the same disk cylinder as the original record, then the use of

trailer records will be more efficient than Lefkovitz's approach whenever

$$A_e(1.5R) < N_k(T_d+T_i)$$

where A_e is the expected number of accesses of this record before the next file reorganization (at which time the record would be pulled together again), and $N_k(T_d+T_i)$ is the one-shot time needed to update the inverted lists when a record is relocated.

For example, in the case of the 20-key records on the 2321 Data Cell, $T_i = P + .5\lceil L/A \rceil 1.5R + 1.5R = 0.725$ sec., $T_d = .069$ sec., $N_k = 20$, and $R = .050$ sec., leading to the result that trailer records will be more efficient whenever the expected number of accesses of that record before the next reorganization is less than 212. When the data base is stored on the 3330 disk, trailer records are more efficient whenever $A_e < 62$. Moreover, even when record trailers are more efficient, it might be desirable to avoid a time-consuming one-time record relocation operation in favor of spending a little extra time on each individual retrieval of that record.

6.2.6 Techniques for Avoiding the Updating of the Inverted List Addresses

In cases where numerous updates cause substantial record expansion and relocation, it might prove effective to have the inverted lists point to positions in an intermediate table rather than directly to the data records. This would tend to enhance the efficiency of Lefkovitz's recommended delete-and-add method of record relocation, since only the single record

address in the intermediate table would have to be updated when a record changed its location; none of the keys would have to have their inverted lists modified. The disadvantage of this approach is that every record retrieval would necessitate one more extra disk accession. Lefkovitz (49) and Bloom (9) have proposed variations on this idea, mostly centering on replacing the record addresses in the inverted lists with the RINs. The actual record address could be determined from each RIN by table look-up or hash coding as appropriate.

If the record addresses in the inverted lists were replaced by RINs, then each access of a record would require an extra disk access (P + 1.5R) to get the record address, and each time the record was relocated the record's address in the RIN table would have to be updated (P + 1.5R + 1.5R) On the other hand, using record addresses in the inverted lists necessitates that when the record is relocated $N_k(T_d + T_i)$ seconds are spent updating the inverted lists of all keys in the record. In general, the RIN approach will be better whenever

$$P_1(P+1.5R) + P_2(P+3.0R) < P_2N_k(T_d+T_i)$$

where P_1 is the probability of a record retrieval and P_2 is the probability of a record relocation. Assuming 20-key records on the 2321 Data Cell, then the RIN approach is superior whenever P_1 is less than 29 times P_2.

6.2.7 Allowing Hierarchical Intra-record Data Relationships

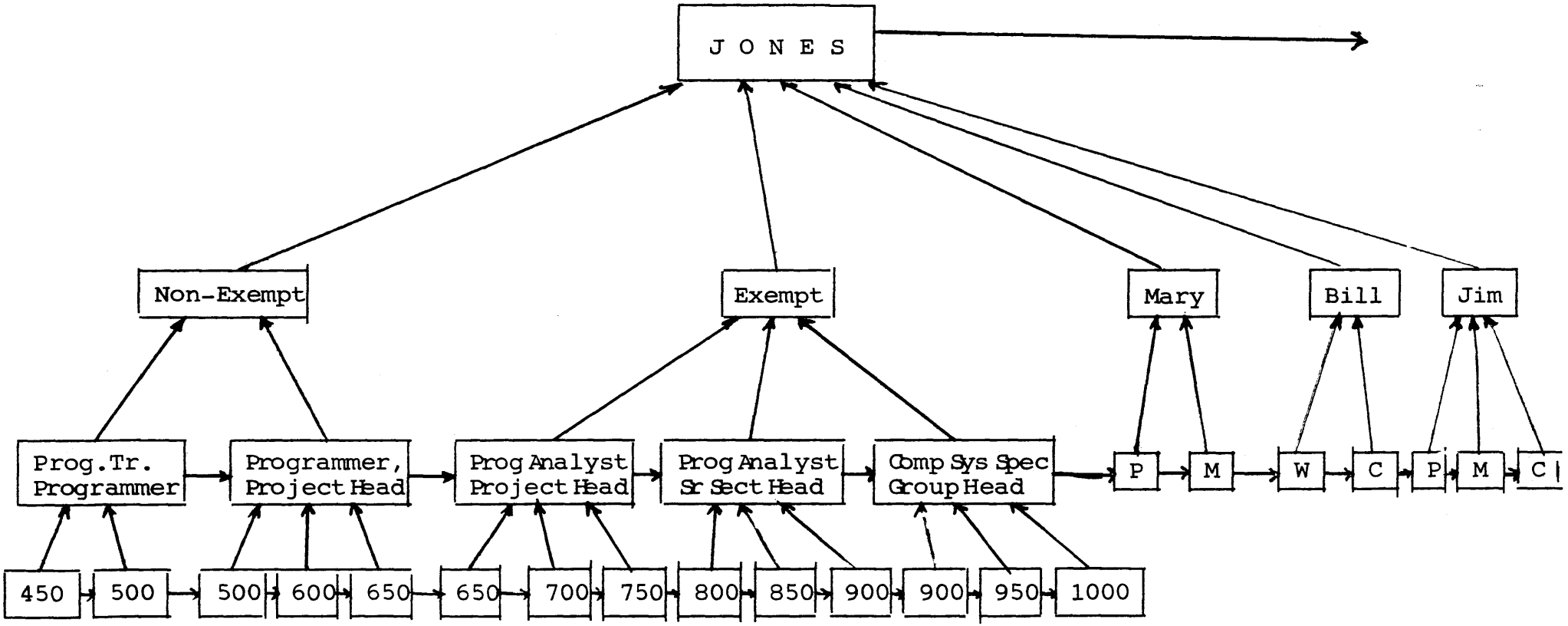
One of the most interesting problems is that of

incorporating hierarchical data associations within the data base records. Lefkovitz (49) has indicated how the system designer can indicate subordinate relationships by creating subrecords within the master record. However, his approach is not practical for a generalized system since any applications programs that operated on the data base would be data dependent in that they would have to have some assumptions concerning record structure built into them.

The Time-Shared Data Management System (TDMS) was designed to handle hierarchical data structures for generalized data bases such that the unsophisticated user need not be aware of the intra-record structure or the overall file organization. The basic approach was described by Bleier (7), and to a lesser degree by Bleier and Vorhaus (8). It consisted of a separate CFIND table for each record that in essence created a doubly-chained hierarchical tree for that record. The CFIND table was described as containing a three-item entry for each node of the tree — (1) a repeating group identifier (RGID); (2) a down-pointer; and (3) an up-pointer. Bleier's description is somewhat confusing until one realizes that he used a slightly non-standard terminology: the CFIND table's so-called up-pointer is really the right pointer of the hierarchical tree; its down-pointer is really an up-pointer of the tree. To better illustrate this mapping, the example that Bleier worked through in table form is shown in tree-structured form in Figure 7.

Winkler (96) and Winkler and Dale (97) described two

Figure 7: Tree Structured Representation of Bleier's TDMS Example



P = Mumps C = Chicken Pox

M = Measles W = Whooping Cough

different Inverted Index systems that allow hierarchically structured records. Both involved the use of a triply-chained tree structure with pointers up, down, and to the right for each node. Winkler's thesis (96) also showed how the hierarchical structure of the record necessitates that the interrelationships between tree nodes governs the scope of the retrieval and maintenance operations. He demonstrated that three general cases covered all possible relationships between the tree node(s) involved in qualifying the scope of the retrieval and the tree node(s) involved in the action to be taken. Typical query examples illustrating this "level of qualification" concept were given by Bleier (7) in his description of the TDMS hierarchical data structures.

6.3 Techniques for Organizing the Inverted Lists

Cardenas (13) has stated that in highly inverted files with many different key values, the managing of the inverted lists is a file problem in itself, possibly of the same magnitude as the organizing of the data base itself. Most authors recommend that the lists themselves be maintained in ordered sequence for efficient logical manipulation.

Lefkovitz (49) has suggested that the inverted lists should be variable length records with reserve space left at the end of each list so that record addresses can be inserted in sequence without overflowing. If the reserved area is exhausted, then the last location of the record contains a link address to another variable length record where the list

continues. When the file is reconstructed, the key lists can be pulled together again into single variable length records. Lefkovitz has also suggested that instead of intersecting an extremely long list with a very short one, it might be more efficient to search the short one directly.

6.3.1 Keeping the Inverted Lists in Order vs. Sorting Them as Needed

Keeping the inverted lists ordered by record address not only speeds up the intersection and merging of lists, but it also keeps the head positioning motions to a minimum in retrieving the records that satisfy single key query. However, to my knowledge no one has quantitatively investigated the trade-off between maintaining the lists in ordered sequence versus sorting them each time a given key appears in a query. In the case of a fast growing data base, where record additions outnumber other on-line operations, it might improve the overall average processing time to reduce the time to add record addresses to the key lists (by inserting them at any convenient location) at the cost of slowing down the retrieval operations. Once the volume of on-line record additions to the data base subsided, the inverted lists would be "permanently" sorted and the system could return to standard operating procedures. Referring back to the example used in Chapter V where the 80-key records and their inverted lists were located on the 2321 data cell, it took 48 seconds (=80 keys x ½ x (4000/250 tracks) x 75 ms/track) to read halfway through each inverted list. If the new key could be added to the last track of the inverted list,

it would have taken only 6 seconds (80 keys x 75 ms/track) to access just the last track of the inverted list. The savings of 42 seconds for each record addition in this case compares very favorably with the time needed to sort in-core each of three lists of 4000 record addresses, a process that would now be required in order to answer a typical three-key query.

In the general case, sorting the inverted lists before each query will be less than the costs of maintaining ordered lists whenever

$$P_1 Q_t S_L + P_2 N_k (P + 3.0R) < P_2 N_k T_i$$

where P_1 is the percentage of on-line operations that are retrievals, P_2 is the percentage that are additions or deletions, and S_L is the time needed to sort a list of length L. Since $T_i = P + .5[L/A]1.5R + 1.5R$, then sorting each time will be most efficient when

$$S_L < P_2 N_k (.5[L/A]-1)1.5R/P_1 Q_t$$

6.3.2 Other Possibilities for Organizing the Inverted Lists

O'Connell (66) has described a small Inverted Index system where the directory and inverted lists are essentially combined into one indexed sequential file. In this system the key directory includes every reference to a given key, not just every unique key; thus, if a given key appeared in 1000 different data records there would be 1000 directory records for it. The advantage of such an implementation is that it is very

easy to implement and maintain using only the manufacturer-provided indexed sequential access method.

Bloom (9) has proposed allowing the inverted lists to be internally organized in one of two different ways. Both are offshoots of his hash coding scheme for decoding the keys. The first has the tracks of the inverted file organized into three components: an "Attribute-Value pair hash area", a ring-structured list of record addresses, and a serially-ordered list of alphanumeric strings. The hash area consists of (1) an attribute code, (2) a pointer to an alphanumeric string for that value, (3) a pointer to the last record address in the ring-structured list, and (4) a pointer to another track if needed. The large number of pointers (each record address in the ring-structured list requires a pointer) involved in this organization implies that it might be overly wasteful of disk storage. Bloom's second approach is to organize in bit-pattern form keys having very long inverted lists. In bit-pattern representation of the inverted lists, each bit position corresponds to a data record, with a 1-bit indicating the presence of the given key in that record, and a 0-bit indicating its absence.

6.3.3 Using Bit-pattern Representation for List Intersection and Merging

Other authors — notably Davis and Lin (24), Spitzer (85), and Siler (83) — have suggested the use of bit-pattern representations, not in the storage of the inverted lists, but in performing the intersecting and merging of the lists to respond

to queries. The Davis and Lin article described a particularly efficient approach to bitmap processing that made use of some of the unique characteristics of their system. Since their data base consisted entirely of fixed length records stored on an IBM 1301 disk at 31 records per track, they used 31 bits of their 36-bit IBM 7090 computer word to designate which records in a given track contained a desired key. To intersect two key lists required only that two arrays containing the bitmap representation of the two keys be logically ANDed word by word. Then if the 19th bit in the 235th word in the resulting array possessed a 1, it meant that the 19th record of the 235th track contained both keys.

6.3.4 Bypassing the Inverted Lists for Unique Key Values

Bleier and Vorhaus (8) reported that when the key value occurred in only one record, TDMS was designed to dispense with the inverted list entirely. In other words, when the list length was one in the output level of the directory, the inverted list pointer pointed directly to the data base record.

6.3.5 Inverted List Alternatives When the Records are Hierarchically Structured

Two papers by Winkler (96) and Winkler and Dale (97) have provided analytical timing comparisons of two Inverted Index systems that differ primarily in the level of detail of the inverted lists. In the first system each inverted list had a pointer to every node in the hierarchical tree (of a record) in which the given key value occurred. Relating this to Figure 7, since the key value "Measles" occurs twice in the

record, there would be two pointers, one to each node. In the second system the inverted list had only one pointer to the start of the record which contained the node, rather than to the node itself. In this case, for the key value "Measles" there would be only one pointer to the beginning of the record. The Winkler and Dale paper which was based on more recent work, gave results which indicated that the first system would generally provide better retrieval times, although the choice between the two systems was not conclusive.

6.3.6 Whether or Not to Provide Inversion on all Queried Keys

One of the most important papers in the Inverted Index area is a 1971 paper by Wang and Lum (91). Among other points they raised, Wang and Lum clearly demonstrated that it is not always most efficient to provide an inverted index on all data fields that appear regularly in queries. By the use of the FOREM file simulation program they showed that for a hypothetical file under consideration, and for one key field that had only four distinct values with relatively even list lengths, it would be more efficient to provide no index on the key field, than to provide one. Under the no index situation, records would be retrieved after they had satisfied the selection criteria of all other key fields, and then would be reviewed in core to see if they also satisfied the conditions on the unindexed key field. They also performed another simulation in which the only thing they changed was the distribution of the key values of this field, skewing it so that one key value was represented in 70% of the records. This time

the FOREM simulation indicated that an implementation with indexes on all key fields would give the best performance. These two results clearly demonstrated the dependence of system performance on data base characteristics and also the need for quantitative evaluation.

CHAPTER VII

TECHNIQUES FOR IMPROVING RETRIEVAL EFFICIENCY BASED ON RECORD CLUSTERING OR MODIFICATION OF THE INVERTED INDEXES

7.1 Explanation of the Problem

The Inverted Index system, even though it generally performs the most efficient retrieval operations of any of the systems described in this paper, still has two principal efficiency limitations: (1) the often substantial overhead it must pay to access and intersect (or merge) the long inverted lists of file addresses; and (2) the fact that the records to be retrieved are often scattered throughout the data base. This latter problem arises chiefly because of the "lumpy" nature of direct access storage devices. The time to access a record on the same track (or cylinder) as the previous record is generally a small fraction of the access time if the records are not located nearby. In fact, the aggregation or clumping together of records that are frequently retrieved together is a desirable enhancement for any of the file organizations discussed so far, as well as for any new data base system that is proposed.

To provide quantitative examples of these two problem areas, consider again the large personnel data base of the company having 50,000 employees. Assume that in order to measure the company's progress in providing equal opportunities for minorities the company wishes to know the names of all

women in the company who are earning $20,000 a year or more. Assume further that the company has 25,000 women employees (mostly secretaries, clerks, and production line workers), around 10,000 people who make over $20,000 a year (engineers, salesmen, managers, etc.), but only 200 women who earn that much (those male chauvinists!). The Multilist organization would first find out that the key list for INCOME>20000 was shorter than that for SEX=FEMALE, and then would proceed merrily through the data base accessing 10,000 records to find the 200 that finally qualified.

The Inverted Index system would be more efficient, but it would nevertheless have to arrive at a medium-sized list of 200 records to be retrieved. Since the number of records to be ultimately retrieved is considerably smaller than the number of records in the data base, few records would be in the same storage buckets, thus leading to a separate disk access for almost every one of the 200 records.

To develop a more accurate statistic on the expected number of disk seeks that would be needed to access a given number or records, a method similar to the one developed by Rothnie (76) in his Ph.D. thesis could be used. Rothnie showed how a Markov State model could be employed to derive the expected number of pages to be retrieved in response to queries, as a function of the file size and the number of records desired. For example, he showed that in a file of 6400 records distributed randomly on 64 pages, the expected number of pages to be retrieved to access 20 records would be 17.3. Although developed for a

paging environment, Rothnie's methodology can be readily transferred to a standard disk environment.

This chapter will report on several methods that have been proposed to deal with the problem of reducing the overhead involved in retrieving and intersecting long key lists (only to wind up with relatively short accession lists), and/or the problem of grouping together records that have similar keys so that disk accesses can be minimized. Neither of these problems is elementary, as witnessed by the fact that many generalized data management systems have avoided tackling either one. The latter problem is particularly imprecise, because a given record may have several keys in common (although different keys) with hundreds of other records, and thus any decision rules for aggregation should define a measure of closeness between records that is based at least partly on the frequency by which given keys appear together in the actual queries and on-line updates.

Some of the pages to be covered here have proposed special procedures to be used as adjuncts and improvements on the standard Inverted Index system. Others have proposed radically new techniques that bring to bear entirely new approaches to record storage and retrieval. Unfortunately, the papers generally share a common limitation in that after reading them, the interested system designer still feels unsatisfied, and uncertain as to their ability to handle the myriad of implementation problems that are part of real-world situations. Questions concerning updating methodology (particularly the question as to what

happens when you modify a given key), concerning the problem of maintaining unequal size clusters on equal-sized tracks, or concerning what happens when some of the restrictive assumptions made to allow mathematically manipulatable equations are relaxed in practice, are often ignored.

Their believability is further restricted by the fact that not one of the papers reports that its system has been fully implemented for a "real world" data base; in fact, few of them have even been tried out on small artificial data bases. And perhaps the most discouraging thing from the point of view of the potential user who would like to compare systems is that each one is described and justified within its own (generally mathematical) framework. Thus, comparing the theoretical performances of the proposed systems is difficult at best, and virtually impossible when the assumptions made contradict one another. With these caveats having been made, let us proceed to discuss the various approaches.

To simplify matters a consistent terminology is used throughout this chapter. The terms "attribute" and "key field", and the terms "attribute value" and "key value" are used interchangeably, and it is assumed, unless otherwise specified, that there are M key fields and N total unique key values. In general, N>>M.

7.2 Techniques for Improving the Inverted Indexing Operations

One obvious technique for reducing the time to generate the fully qualified list of record addresses in multi-key queries

is to provide supplementary inverted indexes that involve two or more key fields (i.e., attributes). Using the example of the previous section, a combined index relating the attributes SEX and SALARY would immediately provide the list of 200 records that satisfy the conjunction of the two key values SEX=FEMALE and SALARY>20000. However, in general, supplementary indexes should be added selectively only for those key fields that occur together frequently in query conjunctions or for which there are only a few key values (e.g., SEX, COLOR OF EYES); otherwise, the storage requirements of the supplementary index files will soon exceed those of the simple inverted indexes.

Lum (54) has proposed doing away with the simple inverted indexes entirely and using only combined index files. Each of his combined index files consists of an ordered M-tuple involving all possible key values of each of the M key fields. Table 16 shows a combined index file for the simple case when there are four key fields (A, B, C, and D) and three key values for each field:

Table 16: A Sample Combined Index File

	M-Tuples	Pointers or RIN's
1.	$A_1B_1C_1D_1$	P_{22}, P_{141}
2.	$A_1B_1C_1D_2$	—
3.	$A_1B_1C_1D_3$	P_{76}
4.	$A_1B_1C_2D_1$	P_8, P_{37}, P_{618}
	⋮	⋮
11.	$A_3B_3C_3D_2$	P_{96}, P_{212}
12.	$A_3B_3C_3D_3$	—

This combined index in Table 16 could immediately point to all records relating to a query involving key field A, key fields A∧B, key fields A∧B∧C, or key fields A∧B∧C∧D. Moreover, all pointers relevant to a particular query are located successively within the index. But what about queries involving key field B∧C or A∧D? Lum showed that in the case of four key fields, only six combined index files are needed to handle all possible cases: ABCD, BCDA, CDAB, DABC, ACBD, BDAC. In the general case $\binom{M}{[M/2]}$ index files are needed for queries involving M key fields.

Since the number of combined index files (and the needed storage) rises rapidly with M, Lum suggested several compromise methods for reducing redundancy at the cost of occasional extra disk access. The principal method involved organizing the combined secondary index files on subgroups of the M key fields. Lum also showed how his system could handle ranges of key values. The main advantages he promoted for the combined index system were its reduced list access time, its elimination of the need to perform key intersections or to search long inverted lists when a given key field had only a few possible values, and its elimination of false drops in cases where the records had complicated internal hierarchical structures.

Mullen (64) followed up on Lum's suggestion to separate the M key fields into subgroups for organizing combined index files to save on file space. Using a simplified analysis, Mullen derived the optimum number of subgroups to construct so as to minimize the average cost of an on-line operation, based

on the weighted sum of retrieval and update operations.

7.3 Systems Involving the Clustering of Similar Records

Ghosh and Abraham (33) proposed in 1967 what they called a Balanced Filing System (BFS), whereby all records having a given set of K unique key values are stored in the same subbucket. Retrieval was based on a complicated system involving deleted finite geometries; to find the subbucket that contained the desired records necessitated that a system of linear algebraic equations be solved. Overall, the authors claimed very low search times for the BFS in comparison with other existing methods.

One of the principal disadvantages of the Balanced Filing System was that it was designed for queries involving exactly K attributes. If fewer than K attributes were specified, additional redundant files were needed. If a range of key values was specified, several accesses were required, generally scattered throughout the file. Moreover, because of the finite geometry approach, the system was difficult to understand, had restricted parameter values, and required the solution of a set of simultaneous algebraic equations in order to obtain the bucket address.

In 1969 Chow (16) proposed a New Balanced File System (NBFS) that he claimed required less storage, provided more useful parameters, and gave even lower retrieval times than BFS. Chow's system computed the bucket address by explicit formula rather than by a set of simultaneous equations and thus was

easier to use, too. However, as was the case with BFS, NBFS was oriented toward retrievals in which the number of attributes in a query was a fixed integer K. Response times for a query involving g attributes ($g \leq K$) were much slower, because several (or many) individual subbuckets had to be accessed. In the general case, if there were N total possible key values, then $\binom{N-g}{K-g}$ subbuckets had to be accessed for a query involving g attributes.

Gustafson (36) in his Ph.D. thesis at the University of South Carolina, and also in a later paper (37), described a randomized combinatorial file structure that bears some basic similarity to the method of Chow. Gustafson assumed that there were N unique key values, and that each record was described by exactly M of them. Thus, there would be exactly $\binom{N}{M}$ possible attribute value combinations, and all records corresponding to a given combination of attribute values would be stored in the same bucket. In general, the number of buckets that had to be accessed in response to a query that specified L attribute values ($1 \leq L \leq M$) was $\binom{N-L}{M-L}$. Thus, if L=M only one bucket had to be retrieved to get all appropriate recores; if L=1 then a large percentage of the buckets usually had to be accessed. A small example showing the number of buckets that must be retrieved when N=14 and M=5 is shown in Table 17.

Note that as the number of unique key values (N) increases, the total number of potential buckets grows astronomically. For example, when N=100 and M=5 the number of possible unique combinations is 7.5×10^7. Because only a small fraction of all

possible bucket combinations would occur in practice, Gustafson suggested the use of a randomizing scheme that would use hash coding to map the many possible attribute value combinations down to a much smaller bucket address space. He also discussed the problem of handling multiple attribute combinations that mapped to the same bucket address. Overall, Gustafson's approach seems to provide high retrieval and updating speed, while requiring minimum storage overhead. However, his assumption that all records be described by exactly M attribute values seems to be overly restrictive for general-purpose applications.

Table 17: Example Showing the Number of Buckets Retrieved Using Gustafson's Approach

N=14 M=5 $\binom{M}{N}$=2002

Number of Attributes Specified (L)	Number of Buckets Retrieved	Percentage Of Buckets Retrieved
1	715 out of 2002	.35714
2	202 out of 2002	.10984
3	55 our of 2002	.02747
4	10 our of 2002	.004995
5	1 out of 2002	.004995

Wong and Chiang (98) have reported on a system that gets around the limitations imposed by others on the number of key values that a given record can have. Under their approach, assuming that there were N unique key values in the system, a

given record could have anywhere from 0 to N keys. Thus, since each key value could either be present or absent in a given record, there are 2^N possible clusters of records. Wong and Chiang termed these clusters "atoms", because they correspond to the irreducible units of a Boolean algebra. Since each record belongs in only one atom, and since the atoms are pari-wise disjoint, intersection of lists need never be performed; since the atoms are totally exhaustive, every retrievable set corresponds to a union of lists of atoms; and finally, since set manipulations are not necessary, neither are inverted lists, and thus all records belonging to a single atom can be stored and retrieved together.

In actual practice only a small percentage of all the 2^N possible atoms would actually exist for a given data base, thus bringing the storage problem down to manageable proportions. However, many other questions remain that were not discussed by Wong and Chiang: How do you quickly find out which of the 2^N possible atoms are specified by a query? How do you go from an atom identification to a bucket address? How do you handle overflow problems since the atoms will vary widely in size? What are the specific procedures for handling key updates —— i.e., moving a record from one atom to another? And what happens to an atom when the last record on it must be moved to another atom? Although the new atom-based system looks very promising, substantial work has to be done before it can be readily applied to real world data bases.

J. B. Rothnie (76) in his 1972 Ph.D. thesis at M.I.T.'s

Civil Engineering Department, and in a later paper with T. Lozano (77), has proposed one of the most useful approaches to record clustering — a method he calls multiple key hashing (mkh). Rothnie assumed that each of the M key fields could take on only one of several distinct key values. The first step in mkh was for the file designer to assign a hashing function h_i to each key field, so as to map the individual key values into (generally) a smaller number of codes. Each record is thus mapped into a characteristic M-tuple based on its M hash code values, and is assigned to a record cluster with all other records that have the same characteristic tuple. If the number of hash codes for key field i is ($nfile_i$), then the total number of clusters (nc)= $\sum_{i=1}^{m} nfile_i$.

Rothnie then showed that for any given combination of key values the appropriate clusters could be retrieved by calculating all characteristic tuples that satisfy the query. For example, a request of the form (KEY_i=VALUE) would require that $nc/nfile_i$ record clusters to be retrieved; a request in which all key fields were assigned values would require the retrieval of just one record cluster.

Rothnie pointed out that for certain key fields (particularly those with small values of $nfile_i$) the number of clusters to be retrieved using mkh could be large, and thus an inverted file approach might be preferred for those attributes. In order to partition the attributes into those that are most efficiently handled by mkh and those that are best handled by

inversion, Rothnie formulated a mathematical program based on minimizing the expected number of disk accesses. He then demonstrated the successful application of a heuristic approximation to the solution on several example files. For these attributes to be organized under mkh, the optimal solution also specified what their individual values of $nfile_i$ should be.

Rothnie's multiple key hashing technique appears to have considerable potential in many real world situations. It appears to be very efficient, easy to use, and to require minimal extra storage. Further documentation is needed concerning file updating and the handling of complex queries in the combined systems where some of the attributes are organized by mkh and some by inversion. The only noticeable restriction of Rothnie's formulation is his assumption that each key field has a single key value. In more complex data bases such key fields as LANGUAGES SPOKEN or CHILDHOOD DISEASES must be able to accept anywhere from zero to a fairly large number of key values for any given record.

CHAPTER VIII

SUMMARY AND RECOMMENDATIONS FOR FUTURE RESEARCH

The principal objective of this thesis has been to demonstrate the feasibility of using a general parameterized model to select the optimum file organization in a given situation. Such a model is designed to predict the expected on-line processing time for different file organizations as a function of the average retrieval time, the average times for various updating operations, and the relative percentage of each type of operation. The average retrieval time and the average updating times are in turn a function of the relevant characteristics of the data base, the queries and requests made by on-line users, and the direct access storage devices used to hold the data base.

A FORTRAN program was developed to demonstrate the usefulness of the model in quasi-"real world" situations. The retrieval and updating characteristics of three well known file organizations — the Multilist, the Inverted Index, and the Cellular Serial — were simulated within the program, and a number of typical cases were run to demonstrate the application of the model to various situations. From these cases several inductively reasoned hypotheses were presented:

(1) When the queries are complex, the Inverted Index file organization performs retrieval operations far more efficiently than either Multilist or Cellular Serial.

(2) When the queries are simple, the Inverted Index and the Multilist file organizations both perform retrieval operations equally well, and both are more efficient than Cellular Serial.

(3) For on-line updating operations, particularly record additions and deletions, the Multilist and Cellular Serial organizations are both more efficient than Inverted Index.

(4) Overall, unless record additions and deletions make up a very high percentage of all on-line operations, the Inverted Index system generally provides lower on-line processing times than either of the other two organizations.

Since the Inverted Index organization proved to be generally superior to the other organizations, a number of specific design alternatives were presented with a view toward optimizing the performance of the Inverted Index system. In each case the strengths and limitations of each alternative were described in order to indicate the appropriate trade-offs between them. Finally, several techniques that have been recently proposed for organizing data bases so as to cluster together similar records were critically reviewed and compared.

Several areas appear to hold considerable promise for future research into the questions of designing superior file organizations and choosing the best one for a given situation:

(1) The use of general parameterized models based on formulas for expected on-line processing time should be pursued. Analytical timing formulas for the retrieval and updating operations of various file organizations should be developed

and incorporated into a general model.

(2) The use of static and dynamic simulation models (such as Senko's FOREM model (91) and Siler's model (83) should be encouraged for comparing the performance of file organizations. Although much more complex than the analytical models, they can provide far more detailed output measures, and their use of probability distributions allows tham to indicate the degree of sensitivity of the system to extreme cases — something that average values usually hide.

(3) Several of the more promising record clustering techniques should be studied further and applied to real data bases. In particular, the methods proposed by Wong and Chiang (98), based on grouping records into the atoms that correspond to Boolean queries, and Rothnie's multiple key hashing system (76,77) are definitely worthy of further research.

(4) The proposals for integrated systems, where different keys are organized according to different file organizations depending on the distribution of their key values and the types of queries in which they are involved, should be pursued. Techniques involving integrated file systems have been proposed by Siler (83) and by Rothnie (76).

(5) Finally, the idea of a generalized information retrieval system, of which only a small part would be the library of undominated file organizations and the file selection model, should be carried further. Another key part of such a system would be a statistical monitoring subsystem that would keep track of the performance of the data base and trigger the file

selection model "to do its thing" at the appropriate times. This monitoring subsystem would have to be sophisticated enough to recognize when deteriorating performance necessitated that the entire data base be restructured using a new file organization, and when it should merely be reorganized under the existing organization to overcome structural inefficiencies caused by numerous record additions and changes. The subsystem would have to operate on both a macroscopic and a microscopic level to keep track of shifts in the types of on-line operations and also to monitor accesses of specific records, keys, and joint combinations of keys to assure optimum performance. Finally, the subsystem will have to be efficient enough so as not to burden the operation of the ongoing information retrieval system.

BIBLIOGRAPHY

1. Angell, T., Automatic Classification as a Storage Strategy for an Information Storage and Retrieval System, Unpublished Masters Thesis, University of Pennsylvania, Philadelphia, (1966).

2. Arora, S.R., and Dent, W.T., "Randomized Binary Search Technique," Communications of the ACM 12, 2 (February 1969), pp. 77-80.

3. Baker, F.T., "Some Storage Techniques for Use with Disk Files," Final Report No. ISR-4, Harvard University, (1963), pp. III-1 through III-43.

4. Bays, C., "The Reallocation of Hash-Coded Tables," Communications of the ACM 16, 1 (January 1973), pp. 11-14.

5. Bell, J.R., "The Quadratic Quotient Method: A Hash Code Eliminating Secondary Clustering," Communications of the ACM 13, 2 (February 1970), pp. 107-109.

6. Bell, J.R., and Kaman, C.H., "Linear Quotient Hash Code," Communications of the ACM 13, 11 (November 1970), pp. 675-677.

7. Bleier, R.E., "Treating Hierarchical Data Structures in the SDC Time-Shared Data Management System (TDMS)," Proceedings of the 22nd ACM National Conference, (1967), pp. 41-49.

8. Bleier, R.E., and Vorhaus, A.H., "File Organization in the SDC Time-Shared Data Management System (TDMS)," Proceedings of the IFIP 1968 Congress, North-Holland Publishing Company, Amsterdam, (1969), pp. 1245-1252.

9. Bloom, B.H., "Some Techniques and Trade-offs Affecting Large Data Base Retrieval Times," Proceedings of the 24th ACM National Conference, (1967), pp. 83-95.

10. Bose, R.C., Abraham, C.T., and Ghosh, S.P., File Organization of Records for Multiple Valued Attributes for Multi-Attribute Queries, IBM Report RC-1886.

11. Brent, R.P., "Reducing the Retrieval Time of Scatter Storage Techniques," Communications of the ACM 16, 2 (February 1973), pp. 105-109.

12. Buchholz, W., "File Organization and Addressing," IBM Systems Journal 2, (June 1963), pp. 86-111.

13. Cardenas, A.F., "Evaluation and Selection of File Organization--A Model and System," Communications of the ACM 16, 9 (September 1973), pp. 540-548.

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

15. Chapin, N., "A Comparison of File Organization Techniques," Proceedings of the 24th ACM National Conference, (1966), pp. 273-283.

16. Chow, D.K., "New Balanced-File Organization Schemes," Informational Control 15, (1969), pp. 377-396.

17. Clampett, H.A., Jr., "Randomized Binary Searching with Tree Structures," Communications of the ACM 7, 3 (March 1964), pp. 163-165.

18. Climenson, W.D., "File Organization and Search Techniques," Annual Review of Science and Technology, Vol. 1, C. Cuadra (Ed.), Wiley, New York, (1966), pp. 107-135.

19. CODASYL Systems Committee, Feature Analysis of Generalized Data Base Management Systems, The Association for Computing Machinery, (May 1971).

20. CODASYL Systems Committee, A Survey of Generalized Data Base Management Systems, The Association for Computing Machinery, (May 1969).

21. Coffman, F.G., and Eve, J., "File Structures Using Hashing Function," Communications of the ACM 12, 7 (July 1970), pp. 427-432,436.

22. Collmeyer, A.J., and Shemer, J.E., "Analysis of Retrieval Performance for Selected File Organization Techniques," Proceedings--1970 Fall Joint Computer Conference, (1970), pp. 201-210.

23. Coyle, F., "The Hidden Speed of ISAM," Datamation, (15 June 1971), pp. 48-49.

24. Davis, D.R., and Lin, A.D., "Secondary Key Retrieval Using an IBM 7090-1301 System," Communications of the ACM 8, 4 (April 1965), pp. 243-246.

25. Day, A.C., "Full Table Quadratic Searching for Scatter Storage," Communications of the ACM 13, 8 (August 1970), pp. 481-482.

26. D'Imperio, M.E., "Data Structures and Their Representation in Storage," Annual Review in Automatic Programming 5, (1969), pp. 1-76.

27. Dodd, G.D., "Elements of Data Management Systems," ACM Computing Surveys 1, 2 (June 1969), pp. 117-133.

28. Flores, I., Data Structure and Management, Prentice-Hall, Englewood Cliffs, New Jersey, (1970).

29. Flores, I., and Madpis, G., "Average Binary Search Length for Dense Ordered Lists." Communications of the ACM 14, 9 (September 1971), pp. 602-603.

30. Foster, C.C., "A Generalization of AVL Trees," Communications of the ACM 16, 8 (August 1973), pp. 513-517.

31. Foster, C.C., "Information Storage and Retrieval Using AVL Trees," Proceedings of the ACM 25th National Conference, (1965), pp. 192-205.

32. Franks, E.W., "A Data Management System for Time-Shared File Processing Using a Cross-Index File and Self-Defining Entries," Proceedings of the 1966 Spring Joint Computer Conference, (1966), pp. 79-86.

33. Ghosh, S.P., and Abraham, C.T., "Application of Finite Geometry in File Organization for Records with Multiple-Valued Attributes," IBM Journal of Research and Development 12, (March 1968), pp. 180-187.

34. Ghosh, S.P., and Ganguly, A.K., Comparison of Techniques for Adding and Organizing Records in the Independent Overflow Area of ISAM, IBM Report RJ-602, (August 1969).

35. Gray, J.C., "Compound Data Structure for Computer Aided Design: A Survey," Proceedings of the 22nd ACM National Conference, (1967), pp. 355-365.

36. Gustafson, R.A., A Randomized Combinatorial File Structure for Storage and Retrieval Systems, Unpublished Ph.D. Dissertation, University of South Carolina, Columbia, South Carolina, (December 1969).

37. Gustafson, R.A., "Elements of the Randomized Combinatorial File Structure," Proceedings of a Symposium on Information Storage and Retrieval, ACM, New York, (April 1971), pp. 163-174.

38. Hibbard, T.N., "Some Combinatorial Properties of Certain Trees with Applications to Searching and Sorting," Journal of the ACM 9, 1 (January 1962), pp. 13-28.

39. Hsiao, D., and Harary, F., "A Formal System for Information Retrieval from Files," Communications of the ACM 13, 2 (February 1970), pp. 67-73.

40. Hsiao, D., and Prywes, N.S., "A System to Manage an Information System," Proceedings of the FID/IFIP Joint Conference on Mechanized Information Storage, Retrieval, and Dissemination, Rome, Italy, (1967), pp. 637-660.

41. Introduction to IBM System/360 Direct Access Storage Devices and Organization Methods, Student Text C20-1649-4, IBM, White Plains, New York, (1969).

42. Johnson, L.R., "An Indirect Method for Addressing on Secondary Keys," Communication of the ACM 4, 5 (May 1961), pp. 218-222.

43. Jones, R.S., "Data File Two--A Data Storage and Retrieval System," Proceedings of the Spring Joint Computer Conference, (1968), pp. 171-181.

44. Knott, G.D., "A Balanced Tree Storage and Retrieval Algorithm," Proceedings of the ACM Symposium on Information Storage and Retrieval, University of Maryland, (1971), pp. 175-195.

45. Knott, G.D., "Expandable Open Addressing Hash Table Storage and Retrieval," 1971 ACM SIGFIDET Workshop - Data Description, Access and Control, (1971), pp. 219-234.

46. Knuth, D.E., The Art of Computer Programming, Vol. 1, Addison Wesley, Reading, Massachusetts, (1968), Chapter 2, pp. 228-463.

47. Lamport, L., "Comment of Bell's Quadratic Quotient Method for Hash Code Searching," Communication of the ACM 13, 9 (September 1970), pp. 571-573.

48. Landauer, W.I., "The Balanced Tree and Its Utilization in Information Retrieval," Transactions on Electronic Computer of the IEEE, Vol. EC-Xii, No. 5, (December 1963), pp. 863-871.

49. Lefkovitz, D., File Structures for On-Line Systems, Spartan Press, Washington, D.C., (1969).

50. Lefkovitz, D., and Powers, R.V., "A List-Structured Chemical Information Retrieval System," Information Retrieval: A Critical View, G. Schecter (Ed.), Thompson Book Company, Washington, D.C., (1967), pp. 109-129.

51. Lowe, T.C., "The Influence of Data Base Characteristics and Usage on Direct Access File Organization," Journal of the ACM 15, 4 (October 1968), pp. 535-548.

52. Luccio, F., "Weighted Increment Linear Search for Scatter Tables," Communication of the ACM 15, 12 (December 1972), pp. 1045-1047.

53. Lum, V.Y., "General Performance Analysis of Key-to-Address Transformation Methods Using an Abstract File Concept," Communications of the ACM 16, 10 (October 1973), pp. 603-612.

54. Lum, V.Y., "Multi-Attribute Retrieval with Combined Indices," Communication of the ACM 13, 11 (November 1970), pp. 660-665.

55. Lum, V.Y., Ling, H., and Senko, M.E., "Analysis of a Complex Data Management Access Method by Simulation Modeling," Proceedings - 1970 Fall Joint Computer Conference, (1970), pp. 211-222.

56. Lum, V.Y., Yuen, P.S.T., and Dodd, N., "Key-to-Address Transform Techniques: A Fundamental Performance Study on Large Existing Formatted Files," Communication of the ACM 14, 4 (April 1971), pp. 228-239.

57. Martin, L.P., "A Model for File Structure Determination for Large On-Line Data Files," Proceedings of the FILE 68 International Seminar on File Organization, (1968), pp. 793-834.

58. Maurer, W.D., "An Improved Hash Code for Scatter Storage," Communications of the ACM 11, 1 (January 1968), pp. 35-38.

59. Meadow, C.T., and Meadow, H.R., "Organization, Maintenance, and Search of Machine Files," Annual Review of Science and Technology, Vol. 5, C. Cuadra (Ed.), Wiley, New York, (1970), pp. 169-191.

60. Mendelson, E., Introduction to Mathematical Logic, P. Van Nostran Company, New York, (1965).

61. Minker, J., and Sable, J., "File Organization and Data Management," Annual Review of Information Science and Technology, Vol. 2, C. Cuadra (Ed.), Wiley, New York, (1967), pp. 123-160.

62. Morris, R., "Scatter Storage Techniques," Communication of the ACM 11, 1 (January 1968), pp. 38-44.

63. Mullin, J.K., "An Improved Index Sequential Access Method Using Hashed Overflow," Communication of the ACM 15, 5 (May 1972), pp. 301-307.

64. Mullin, J.K., "Retrieval-Update Speed Tradeoffs Using Combined Indices," Communication of the ACM 14, 12 (December 1971), pp. 775-776.

65. Nijssen, G.M., "Efficient Batch Updating of a Random File," Proceedings - 1971 ACM SIGFIDET Workship - Data Description, Access, and Control, (November 1971), pp. 173-186.

66. O'Connell, M., "A File Organization Using Multiple Keys," Spring Joint Computer Conference, (1971), pp. 539-544.

67. Patt, Y.N., "Variable Length Tree Structures Having Minimum Average Search Time," Communication of the ACM 12, 2 (February 1969), pp. 72-76.

68. Perlis, A.J., and Thornton, C., "Symbol Manipulation by Threaded Lists," Communication of the ACM3, 4 (April 1960), pp. 195-204.

69. Peterson, W.W., "Addressing for Random-Access Storage," IBM Journal of Research and Development 1, (1957), pp. 130-146.

70. Price, C.E., "Table Lookup Techniques," ACM Computing Survey 3, (June 1971), pp. 49-65.

71. Prywes, N.S., "Man-Computer Problem Solving with Multilist," Proceedings IEEE 54, 12 (December 1966), pp. 1788-1801.

72. Prywes, N.S., and Gray, H.J., "The Multi-List System for Real-Time Storage and Retrieval," Proceedings of the IFIP Congress 1962, Munich, Germany, (September 1962), pp. 273-278.

73. Prywes, N.S., and Gray, H.J., "The Organization of a Multi-list-type Associative Memory," Proceedings of the Session on Gigacycle Computing Systems, AIEE, Publication S-136, (January 1962), pp. 87-101.

74. Radke, C.E., "The Use of Quadratic Residue Research," Communication of the ACM 13, 2 (February 1970), pp. 103-105.

75. Rettenmayer, J.W., The Effect of File Ordering on Retrieval Cost, Unpublished Ph.D. Dissertation, University of California, Los Angeles, (1969).

76. Rothnie, J.B., The Design of Generalized Data Management Systems, Unpublished Ph.D. Dissertation, Massachusetts Institute of Technology, Cambridge, Massachusetts (September 1972).

77. Rothnie, J.B., and Lozano, T., "Attribute Based File Organization in a Paged Memory Environment," Communication of the ACM 17, 2 (February 1974), pp. 63-69.

78. Rotwitt, T., and DeMaine, P.A.D., "Storage Optimization of Tree Structured Files Representing Descriptor Sets," 1971 ACM SIGFIDET Workshop - Data Description, Access, and Control, (November 1971), pp. 207-217.

79. Sagamang, J.P., Automatic Selection of Storage Structure in a Generalized Data Management System, Unpublished Masters Thesis, University of California, Los Angeles, (June 1971).

80. Senko, M.E., "File Organization and Management Information Systems," Annual Review of Information Science and Technology, Vol. 4, C. Cuadra (Ed.), Wiley, New York, (1969), pp. 111-141.

81. Shneiderman, B., "Optimum Data Base Re-organization Points," Communication of the ACM 16, 6 (June 1973), pp. 362-365.

82. Shoffner, R.M., "The Organization Maintenance, and Search of Machine File," Annual Review of Information Science and Technology, Vol. 3, C. Cuadra (Ed.), Wiley, New York, (1968), pp. 137-167.

83. Siler, K.F., A Stochastic Model for the Evaluation of Large Scale Data Retrieval Systems: An Analysis of Data Base Inversion and Key Truncations, Unpublished Ph.D. Dissertation, University of California, Los Angeles, (December 1971).

84. Skidmore, A.K., and Weinberg, B.L., "Storage and Search Properties of a Tree-Organized Memory System," Communication of the ACM 6, 1 (January 1963), pp. 28-31.

85. Spitzer, J.H., "Storing the Directory for an Inverted List System," Data Base 1, 4 (Winter, 1969), pp. 12-14.

86. Stone, H.S., Introduction to Computer Organization and Data Structures, McGraw-Hill, New York, (1972).

87. Sussenguth, E.H., "Use of Tree Structures for Processing Files," Communication of the ACM 6, 5 (May 1963), pp. 272-279.

88. Systems Development Corporation, DS-2 User Manual, SDC Corporation, Santa Monica, California, (1970).

89. Taylor, A., "ISAM Overhead and Its Replacement: AMIGOS," Computer World, (5 Editorials), 4 November 1970 through 6 January 1971.

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

91. Wang, C.P., and Lum, V.Y., "Quantitative Evaluation of Design Tradeoffs in File Systems," Proceedings of a Symposium on Information Storage and Retrieval, Association of Computing Machinery, New York, (April 1971), pp. 155-162.

92. Weinberg, P.R., A Time Sharing Chemical Information Retrieval System, Unpublished PH.D. Dissertation, University of Pennsylvania, Philadelphia, (1968).

93. Weizenbaum, J., "Knotted List Structures," Communication of the ACM 5, 3 (March 1962), pp. 161-165.

94. Wexelblat, R.L., and Freedman, H.A., "The MULTILANG On-Line Programming System," Proceedings of the AFIPS 1967 Spring Joint Computer Conference, Vol. 30, Thompson Book Company, Washington, D.C., pp. 559-569.

95. Williams, W.D., and Bartram, P.R., "COMPOSE/PRODUCE: A User-Oriented Report Generator Capability within the SDC Time Shared Data Management System," Spring Joint Computer Conference, (1967), pp. 635-640.

96. Winkler, A.J., A Methodology for Comparison of File Organization and Processing Procedures for Hierarchical Storage Structures, Unpublished Ph.D. Dissertation, The University of Texas, Austin, (May 1970).

97. Winkler, A.J., and Dale, A.G., "File Structure Determination," Proceedings of the Sumposium on Information Storage and Retrieval, University of Maryland, College Park, Maryland, (April 1971), pp. 133-146.

98. Wong, E., and Chiang, T.C., "Canonical Structure in Attribute Based File Organization," Communication of the ACM 14, 9 (September 1971), pp. 593-597.

APPENDIX A

LISTING OF THE FORTRAN PROGRAM USED FOR CALCULATIONS BASED ON THE RETRIEVAL AND UPDATING FORMULAS

```
00100 PROGRAM DECRUL(INPUT,OUTPUT,TAPE6=OUTPUT)
00110C
00120C    THIS PROGRAM IS DESIGNED TO CALCULATE THE ESTIMATED
00130C      RETRIEVAL TIMES AND UPDATING TIMES (OF VARIOUS
00140C     TYPES) FOR 3 DIFFERENT FILE ORGANIZATIONS: MULTI-
00150C     LIST, INVERTED INDEX, AND CELLULAR SERIAL. THE
00160C     DATA BASE PARAMETERS, DASD SPECIFICATIONS AND
00170C     QUERY CHARACTERISTICS ARE INCLUDED AS DATA WITHIN
00180C     THE PROGRAM.
00190C
00200 REAL  IL1(25),ML1(25),CS1(25)
00210 REAL  IL2(5),IL3(5),IL4(5),IL6(5),IL5(5),IL7(5)
00220 REAL  ML2(5),ML3(5),ML4(5),ML5(5),ML6(5),ML7(5)
00230 REAL  CS2(5),CS3(5),CS4(5),CS5(5),CS6(5),CS7(5)
00240 REAL  P1(10),P2(10),P3(10),P4(10)
00250 INTEGER  V,RQ,I1(25),I2(25),I3(25)
00260 DATA  NTESTS / 13 /
00270 DATA  I1(1),I2(1),I3(1) / 5,50,0 /
00280 DATA  I1(2),I2(2),I3(2) / 5,50,1 /
00290 DATA  I1(3),I2(3),I3(3) / 5,50,5 /
00300 DATA  I1(4),I2(4),I3(4) / 5,50,25 /
00310 DATA I1(5),I2(5),I3(5) / 5,50,45 /
00320 DATA  I1(6),I2(6),I3(6) / 5,200,25 /
00330 DATA  I1(7),I2(7),I3(7) / 20,200,25 /
00340 DATA  I1(8),I2(8),I3(8) / 20,200,100 /
00350 DATA  I1(9),I2(9),I3(9) / 20,800,25 /
00360 DATA  I1(10),I2(10),I3(10) / 20,800,100 /
00370 DATA  I1(11),I2(11),I3(11) / 80,800,100 /
00380 DATA  I1(12),I2(12),I3(12) /80,800,600 /
00390 DATA I1(13),I2(13),I3(13) / 80,3500,600 /
00400 DATA  P1(1),P2(1),P3(1),P4(1) / .90,.03,.03,.04 /
00410 DATA P1(2),P2(2),P3(2),P4(2) / .60,.05,.05,.30 /
00420 DATA P1(3),P2(3),P3(3),P4(3) / .50,.20,.20,.10 /
00430 DATA P1(4),P2(4),P3(4),P4(4) / .10,.10,.10,.70 /
00440 DATA P1(5),P2(5),P3(5),P4(5) / .10,.50,.30,.10 /
00450C
00460 PRINT,*IF DATA BASE FILES ARE ON A 2321 DATA CELL INPUT A "1"*
00470 PRINT,*IF THEY ARE ON A 3330 DISK INPUT A "2"*,
00480 READ,MM
00490 V = 10000
00500 NR = 500000
00510 CF = 2000
00520 IF(MM.EQ.1)  RC=100
00530 IF(MM.EQ.2)  RC=114
00540 QT = 4
00550 QN = 3
00560 AFAST = 1600
```

```
00570 IF(MM.EQ.1)ASLOW=250
00580 IF(MM.EQ.2)  ASLOW=1600
00590 PFAST = 27
00600 IF(MM.EQ.1)  PSLOW=500
00610 IF(MM.EQ.2)  PSLOW=27
00620 IF(MM.EQ.1)  RTSLOW=55
00630 IF(MM.EQ.2)  RTSLOW=806
00640 RFAST = 16.7
00650 IF(MM.EQ.1)  RSLOW=50
00660 IF(MM.EQ.2)  RSLOW=16.7
00670 TDIR = PFAST + 2.5*RFAST
00680 TTFAST = PFAST + 1.5*RFAST
00690 TTSLOW = PSLOW + 1.5*RSLOW
00700 TACCES = PSLOW + .5*RSLOW + CF/RTSLOW
00710C
00720C     CALCULATE THE RETRIEVAL TIMES
00730C
00740 TCELL = PSLOW + RC*CF/RTSLOW
00750 WRITE(6,21) TDIR,TTFAST,TTSLOW
00752 21 FORMAT(//*TIME TO READ THE DIRECTORY*,-3PF9.3 /
00753+          *TIME TO READ A TRACK(FAST)*,-3PF9.3 /
00754+          *TIME TO READ A TRACK(SLOW)*,-3PF9.3 )
00756 WRITE(6,22) TACCES,TCELL
00757 22 FORMAT(*TIME TO ACCESS A DATA RECORD*,-3PF7.3 /
00758+          *TIME TO READ A FULL CELL*,-3PF11.3//)
00760 WRITE(6,25)
00770 25 FORMAT(*SUMMARY OF RETRIEVAL TIMES UNDER VARIOUS *,
00780+   *CONDITIONS*)
00790 WRITE(6,30)
00800 30 FORMAT(/6X,*NUM    LIST  S-LIST   QUERY   INVRTD   MU*,
00810+  *LTI-    CELL.* /5X,*KEYS  LENGTH  LENGTH   RESP.    INDEX*,
00820+  *     LIST   SERIAL*)
00830C
00840 DO 100  I=1,NTESTS
00850 NK = I1(I)
00860 LS = I2(I)
00870 RQ = I3(I)
00880 L = NR*NK/V
00890 CK = MIN0(L,NR/RC)
00900 INUM1 = INT(L/ASLOW+.9999)
00910 INUM2 = INT(CK/AFAST+.9999)
00920 IL1(I) = QT*TDIR + QT*(PSLOW+1.5*INUM1*RSLOW) + RQ*TACCES
00930 ML1(I) = QN*TDIR + LS*TACCES
00940 CS1(I) = QN*TDIR + QN*(PFAST+1.5*INUM2*RFAST) +
00950+        RQ*(PSLOW + RC*CF/RTSLOW)
00960 WRITE(6,51)  I,NK,L,LS,RQ,IL1(I),ML1(I),CS1(I)
00970 51 FORMAT(I3,*.*,I5,3I8,-3P3F9.1)
00980 100 CONTINUE
00990C
01000C     CALCULATE THE UPDATE TIMES
01010C
01020 N = 2
01030 F = .2
```

```
01040 TUPDIR = TDIR + 1.5*RFAST
01050 WRITE(6,200) F
01060 200 FORMAT(/////*SUMMARY OF UPDATE TIMES    F =*,F3.1)
01070 WRITE(6,220)
01080 220 FORMAT(/*  NUM    LIST   INVRTD   MULTI-    CELL.*/
01090+  * KEYS  LENGTH    INDEX     LIST   SERIAL*)
01100 DO 300  I=1,3
01110 NK = 5*(4**(I-1))
01120 L =NR*NK/V
01130 CK = MIN0(L,NR/RC)
01140 INUM1 = INT(L/ASLOW+.9999)
01150 INUM2 = INT(CK/AFAST+.9999)
01160 TINV = PSLOW + .75*RSLOW*INUM1 + 1.5*RSLOW
01170 IF (INUM1.EQ.1)    TINV = PSLOW + 3.0*RSLOW
01180 TCS = PFAST + .75*RFAST*INUM2 + 1.5*RFAST*F
01190 IF (INUM2.EQ.1)    TCS = PFAST + 1.5*RFAST + 1.5*RFAST*F
01200 IL2(I) =          TACCES + NK*(TUPDIR+TINV)
01210 IL3(I) = TDIR + 2*TACCES + NK*(TUPDIR+TINV)
01220 IL4(I) = TDIR + 2*TACCES
01230 IL5(I) = TDIR + 3*TACCES + NK*(TDIR+TINV)
01240 IL6(I) = TDIR + 2*TACCES + N*(TUPDIR+TINV)
01250 IL7(I) = TDIR + 2*TACCES + N*(TUPDIR+TINV)
01260C
01270 ML2(I) =          TACCES + NK*TUPDIR
01280 ML3(I) = TDIR + 2*TACCES
01290 ML4(I) = TDIR + 2*TACCES
01300 ML5(I) = TDIR + 3*TACCES + NK*TUPDIR
01310 ML6(I) = TDIR + 2*TACCES
01320 ML7(I) = TDIR + 2*TACCES + N*TUPDIR
01330C
01340 CS2(I) =          TACCES + NK*(TDIR+TCS)
01350 CS3(I) = TDIR + 2*TACCES
01360 CS4(I) = TDIR + 2*TACCES
01370 CS5(I) = TDIR + 2*TACCES
01380 CS6(I) = TDIR + 2*TACCES
01390 CS7(I) = TDIR + 2*TACCES + N*(TDIR+TCS)
01400 WRITE(6,302)  NK,L,IL2(I),ML2(I),CS2(I),IL3(I),ML3(I),
01410+   CS3(I),IL4(I),ML4(I),CS4(I),IL5(I),ML5(I),CS5(I),
01420+   IL6(I),ML6(I),CS6(I),N,IL7(I),ML7(I),CS7(I),N
01430 302 FORMAT(I5,I8,-3P3F9.1,*  REC.ADDITION*/
01440+   13X,-3P3F9.1,*  REC.DELETION* / 13X,-3P3F9.1,
01450+  *  MOD W/O RELOC.*/13X,-3P3F9.1,*  MOD WITH RELOC.*/
01460+   13X,-3P3F9.1,*  DELETE*,I2,* KEYS* / 13X,-3P3F9.1,
01470+   *  ADD*,I2,* KEYS* /)
01480 300 CONTINUE
01490C
01500C     RUN THROUGH THE DECISION RULES FOR SEVERAL TEST CASES
01510C
01520 WRITE(6,502)
01530 502 FORMAT(/////*SOME EXAMPLES USING THE BASIC *
01540+   *DECISION RULES*)
01550 DO 500  I=1,3
01560 IF (I.EQ.1)   K=3
```

```
01570 IF (I.EQ.2)   K=7
01580 IF (I.EQ.3)  K=13
01590 WRITE(6,504) I,I1(K),I2(K),I3(K)
01600 504 FORMAT(// *CASE*,I2,*    NK =*,I4,*    LS =*,I4,
01610+   *    QR =*,I4 // 23X, *INVRTD INDEX*,5X,*MULTI-LIST*,
01620+   5X,*CELL. SERIAL*/ *   P1   P2   P3   P4*,
01630+   3(*  RETRVL  OVRALL*))
01640 DO 500  J=1,5
01650 X1 = P1(J)*IL1(K) + P2(J)*IL2(I) + P3(J)*IL3(I)
01660+      + P4(J)*IL4(I)
01670 X2 = P1(J)*ML1(K) + P2(J)*ML2(I) + P3(J)*ML3(I)
01680+     + P4(J)*ML4(I)
01690 X3 = P1(J)*CS1(K) + P2(J)*CS2(I) + P3(J)*CS3(I)
01700 X4 =      + P4(J)*CS4(I)
01710 WRITE(6,506) P1(J),P2(J),P3(J),P4(J),IL1(K),X1,ML1(K),
01720+   X2,CS1(K),X3
01730 506 FORMAT(4F5.2,-3P6F8.1)
01740 500 CONTINUE
01750 PRINT,* *////
01760 CALL EXIT
01770 END
```

APPENDIX B

SUSPECTED NUMERICAL ERRORS IN THE PAPERS BY LEFKOVITZ(49) AND MARTIN(57)

Two of the primary sources for this paper — the book, File Structures for On-Line Systems by D. Lefkovitz and a paper by L. D. Martin on the FILE 68 International Seminar on File Organization — both seem to contain several errors in analysis and calculation. This is unfortunate since these are among the only sources that present side-by-side timing analyses.

First of all, it should be mentioned that their two sets of timing formulas differ from each other (and from mine) particularly in the area of update timing. Some of these differences can be traced back to differing assumptions, and some to the fact that Lefkovitz computes overall processing times, whereas Martin works with processing time differences. However, a residual set of formulas remain that appear to be irreconcilable. My own conclusions as to appropriate update timing estimates and their justifications are presented in Chapter V.

This Appendix deals with some of the suspected errors in Martin's and Lefkovitz's numerical examples. These examples are important in that they give the reader an intuitive feeling for the efficiency of the various techniques; errors in the numerical results can thus have a particularly confusing effect

on the unsuspecting reader.

Martin tried out his model on a total of 625 ($=5^4$) test cases, with five different values for each of four different parameters (N_k, L_s, Q_r, and P) being used in all possible permatations. He published the results of seventeen of these test cases; in five of them Inverted Index was best; in seven, Multilist was best; in five, Cellular Serial was best. However, he made two critical errors that negate much of his sample results.

First of all, in all seven cases in which Multilist proved to be best, L_s (the shortest list length) was less than Q_r (the query response). This is physically impossible, since by definition for a conjunction of terms Q_r L_s in all cases.

Secondly, Martin incorrectly gave the transfer rate of strip memory for the IBM 2321 Data Cell as 312,000 char/sec (it is actually 55,000). Since the transfers of entire cells to core is by far the dominant part of retrieval operations for the Cellular Serial organization, the overall timing for this system may have been seriously underestimated. This casts doubts on the five published cases where Cellular Serial turned out best. Because of Martin's choice of inaccurate and infeasible parameter values, much of his test results are incorrect and confusing.

Lefkovitz's analysis seems to be more thorough; however, he too made some numerical mistakes that led to misleading results. On Page 176 of his book, he presented a sample case

in which the retrieval time for the Cellular Serial system was calculated at 3.6 seconds. Based on his timing formulas and his particular parameter values, I compute the retrieval time for Cellular Serial as being .9 + Q_r(4.5) seconds. Thus, even if only one query response is being retrieved, Cellular Serial should take at least 5.4 seconds; if 15 queries are being retrieved, it takes 68.4 seconds.

Moreover, on the very same page he stated that the update time for non-key modification with relocation for the Inverted Index system is 25.8 seconds. I calculate it at 15.8 seconds. Lefkovitz appears to be in error, because he calculated the time for Inverted Index whole record deletion as being 15.8 seconds, and two pages earlier derived that for Inverted Index the modification of non-key data with relocation and the deletion of entire records have the same timing formulas.

Other differences between my numerical results and those of Lefkovitz can be attributed to different assumptions and to my revisions to his timing formulas as described in Chapter V.