DIRECTORY DESIGN AND RECORD ALLOCATION
FOR LIST AND CLUSTER FILES*

Chung-Shu Hubert Yang

TR 76-272

March 1976

Department of Computer Science
Cornell University
Ithaca, New York 14853

*This research was supported in part by the National Science Foundation under grant GJ-43505.

BIOGRAPHICAL SKETCH

Chung-Shu Hubert Yang was born on June 26, 1948 in Nanking, China. In May 1970, he received the B.S. degree in Electrical Engineering from National Taiwan University. He served in the army for a year before he entered the graduate school at Cornell University in September 1971. He obtained an M.S. in Computer Science in 1973.

To my parents, Cheng-Ching and Chu-Zu

ACKNOWLEDGMENTS

I am deeply indebted to my thesis advisor, Professor Gerard Salton, for his continued guidance and encouragement not only in the preparation of this thesis but throughout my four years of study at Cornell.

Thanks are also due to Professors John E. Dennis, Jr., Lionel Weiss, and Shih-Ping Han, for serving on my special committee.

I thank my wife, Ming-Whui, whose love, patience, and confidence continuously supported me and my work.

TABLE OF CONTENTS

CHAPTER | PAGE

DIRECTORY DESIGN AND RECORD ALLOCATION
FOR LIST AND CLUSTER FILES

Chung-Shu Hubert Yang, Ph.D.
Cornell University 1976

Most file organizations for on-line secondary key retrieval consist of two subcomponents - a structural file and a data file. The structural file provides, a physical access path in the data base for each query, so that searches can be restricted to small portions of the data file. The data file contains the data records where information is stored. Hence, the file design problem consists of (1) an efficient design of the structural file, and (2) an allocation of data records in the data file so that a given set of data records can be jointly retrieved at minimum cost. List structure organizations and clustered organizations are shown to be important structures for secondary key retrieval. This thesis studies the design aspects of the structural and data files for both of the above organizations.

The most popular list structure files are the inverted list and the multilist organizations. It is shown that either organization is a special case of a new class of hybrid list organizations. File elements in this class are characterized by a list length parameter ℓ_{th}, and a list is stored as an inverted list or as a multilist depending on whether its length is larger than ℓ_{th} or not. Analytical and simulation

results indicate that neither a pure inverted list organization nor a pure multilist organization is normally the best choice for all elements in the class.

A new method is also introduced for the structure of combined indices for list structure files. A combined index is created only if its component keys co-occur frequently in the queries. Experimental results show that such combined indices do improve the search performances for both inverted list and multilist organizations.

Search methods and physical implementations of clustered organizations are discussed. The balancing of cluster trees is shown to be an important concept. A search model is established to obtain optimal branching ratios for cluster trees. The optimal branching ratio is seen to be dependent on certain statistical characteristics of the data base.

The last part of the thesis concerns itself with the arrangement of records in the data file. The data records are assigned to blocks in the disk like devices. The goal is to minimize the average number of block accesses in processing the query set in the system. This problem turns out to be a polynomial complete problem. Heuristic algorithms are therefore used for the record block assignment. Experimental results show that a record organization produced by the heuristic algorithms is more efficient than a random assignment of records to the blocks.

CHAPTER 1 INTRODUCTION

In contrast to those scientific applications where complicated and prolonged arithmetic and logical operations are performed on relatively small amounts of data, information processing systems typically handle very large volumes of data using relatively trivial computations. In the present state of computer technology, it is not possible to store in main memory even a small portion of the data base used in an information system. The data are therefore placed on backup storage devices (disks, drums, tapes, etc.), and only small parts are brought to the main memory during the processing of a query. Unfortunately the speed of data transfer between backup storage and main memory is approximately three orders of magnitude slower than the computational speed in the main memory. It is therefore important to transfer to the main memory only the data which are pertinent to a query.

Two approaches are often used to solve this problem. First, all queries submitted to the system within a period of time may be collected and processed in a batch; this requires a serial scan of every record in the file. If the number of queries processed together as a batch is large enough, it is likely that most records in the file will be pertinent to at least one query. Hence, one can apportion the cost of the serial scan of the file by dividing this among all the queries in a batch. A second approach is to process a query as soon as it is received by the system. In such circumstances, the system must restrict the search scope to a small portion of the file. This can be achieved by designing sophisticated file organizations which supply special access paths to the records. A serial scan of the file is thus avoided.

Both approaches are valuable for information retrieval systems. For example, in a credit account processing environment, the customers are billed once a month. It is thus reasonable to process in a batch all the transactions which are collected in a month. For some on-line system, say an airline reservation system, it is necessary that each query be processed in real time. The second approach must then be adopted by searching a small portion of the file for each query.

The queries processed in an information system can generally be classified into two types : primary key retrieval queries and secondary key retrieval queries (Knuth 1973). In primary key retrieval the unique record which is identified by the key specified in the query must be retrieved. In secondary key retrieval all the records in the file that satisfy a condition composed of one or more keys in the query must be obtained. The purpose of this thesis is the study of certain file design problems arising in a real time information system for secondary key retrieval. The goal is to reduce the number of records which must be examined in order to produce answers to a query. An attempt is also made to assign to convenient locations in storage the set of records that must be examined. Such records should then be jointly retrievable with as little effort as possible.

Chapter two of the thesis first reviews two typical kinds of information systems, the attribute-based systems and the keyword-based systems. It is shown that, although the two types of systems are different in the scope of application, record formatting, query specification, and retrieval constraints, many file organization techniques are applicable to both types of information systems. A generalized schematic diagram is then introduced for file organizations and the associated search procedures. The generalized file organization consists of a structural file and a data file. The structural file provides, for each query, a physical access path in the data file so that searches can be limited to small portions of the data file. Under such a framework, five principal file organizations -- the sequential file organization, the indexed sequential file organization, the scattered storage file organization, the list structure file organization and the clustered file organization -- are reviewed. Each of these is shown to consist of a special realization of the generalized file organization schema. List file organizations and clustered file organizations are shown to be mainly used for on-line processing of secondary key retrievals. In chapters three, four and five techniques are proposed to improve the structural file search for list structure organizations and clustered organizations. Chapter six concerns itself with the fast retrieval of records from the data file following the structural file search. The contents of the four chapters are summarized briefly in the following paragraphs.

A Structure and Search Algorithm for a Class of List File Organizations (Chapter 3)

Querying and updating procedures are analyzed for an inverted list organization and a multilist organization. An algorithm is proposed for retrieving exactly once all the records which are included in the union of more than one unsorted multilist. Querying costs using multilists and inverted lists are then compared. A new class of hybrid list organizations is introduced as a result of the comparison. A hybrid list organization of parameter l_{th} is a list structure which stores the set of identifiers for the records containing key K_i as a separate inverted list if there are more than l_{th} records in which K_i appears; if there are fewer than l_{th} records for K_i, the set of identifiers is embedded in the data records as a multilist organization. When the values of parameter l_{th} vary from zero infinity, a whole class of hybrid list organizations is obtained.

The pure multilist and pure inverted list organizations are included as two special elements in the class. A simplified system model is adopted to show analytically that one can indeed select some instances of the hybrid list organization which outperform both the inverted list organizations and the multilist orgnaizations. Extensive simulation results for more realistic models are also obtained. The simulation results reach conslusions consistent with those obtained in the simplified analytical model.

<u>Combined Indices in List Structure File Organizations (Chapter 4)</u>

List structure organizations are based on the notion of creating linkages for a set of data records - either by the use of an inverted list or by the use of a multilist. The records included in a given list are characterized by a specific key attached to each of the records. The criterion of linking the records need not necessarily be restricted to the inclusion of a single key only. If the joint existence of two or more keys is frequently specified by the queries, it is probably advisable to create combined lists such that each list links the set of records containing several keys simultaneously. The index to a combined list can be called a "combined index". Certain previous approaches to the creation of combined indices are reviewed, The combined indices created by these methods facilitate very fast query retrieval but result in complicated file maintenance operations and excessive storage requirements. A new method is introduced for the structure of combined indices consisting of the keys that co-occur frequently in the queries. In an inverted list organization, the use of such combined indices reduces querying costs as well as updating costs while the storage space needed is essentially the same as when the combined indices are not used. When the combined indices created by the new method are used in multilist organizations, the query processing is speeded up considerably. Some update operations are a bit harder to perform while others become easier. A small amount of additional storage is necessary. However, if combined indices are constructed only for those keys that are frequently used, the additional storage requirement is very small. Experimental results show that heuristically created combined indices do improve the querying costs for both inverted list and multilist file organizations.

Clustered File Organization (Chapter 5)

In addition to the list structure file organizations, clustered file organizations are also useful for on-line secondary key retrieval. The clustered file organization partitions the data file into groups of related records in the hope that the relevant records for any query can be found in a small number of groups, and that few irrelevant data records are contained in such groups. The structure of a clustered file consists of a cluster tree serving as a search directory, and a set of data records. The search methods and physical implementations for a clustered file organization are discussed. The balancing of cluster trees is shown to be valuable for the rapid generation and efficient retrieval of records in clustered file organizations. A search model is proposed to assess optimal branching ratios for cluster trees. The model is programmed and tested on several sample files. The optimal branching ratio is shown to be dependent on certain statistical characteristics of the processing environment - for example, the number of data records in the file, and the number of data records retrieved by a query.

Optimal Record Address Allocation Schemes (Chapter 6)

Chapter six deals with the allocation of record addresses in the data file. It is assumed that a subset of the data records has already been designated for retrieval using some the structural files. The problem of interest is then to arrange the records in the data file in such a way that any subset of data records which is jointly retrieved by a query can be obtained as fast as possible. The consecutive retrieval property of record address assignment proposed by Ghosh (1972) is examined. A query is said to possess the consecutive retrieval property if all its relevant records are stored consecutively in the backup storage device. Such a property is seldom true simultaneously for a whole set of queries. Furthermore, the consecutive retrieval property is shown to be an unrealistic view of record address allocation in a multiprogramming computer system. Alternative views of the record address assignment, the ideal retrieval property and the minimal access retrieval property, are proposed. In each case, the aim is to minimize the number of disk blocks in which the relevant records of any query are located. The problem of assigning records into blocks so as to achieve the minimal access retrieval property for a set of queries is shown to be polynomial complete. Therefore, heuristic algorithms are used for the record block assignment. Experimental results show that a record organization produced by such heuristic algorithms is more efficient than a random assignment of records to the storage blocks.

It is concluded that all methods treated are realistic and implementable in operational retrieval situations.

Chapter 2 A Survey of File Organization Techniques in Information Systems

2.1 Introduction

The design of general purpose information systems has received increased attention in both universities and industries in the past few years. The design aspects that draw most interest include models of information system, user interface to the system, integrity and security, file organization, indexing theory, etc.. This chapter first discusses two major types of information systems that are presently in operation in many institutions, and then reviews the important file organization techniques that support these information systems.

2.2 Levels of Data Structure

Within a data base there exist several levels of data. As pointed out by Olle (1969), the three principal levels in the data hierachy are file level, record level, and field level. A data base consists of one or more files. The files are linked if a relation exists among them or unlinked if otherwise. A file is a set of records. And the records in a file correspond to occurrences of a single record type that describes a specific class of real world objects. For example, a personnel file is a collection of record occurrences of the employee record type which is designed to store information about employees.

A record is further subdivided into a number of fields. The fields can in general be classified into two kinds, Key fields and data fields. The contents of key fields (called keys) are used to distinguish one record from another and they can be used to specify the retrieval criteria in queries. In any given file, each record should be uniquely designated by an identifier which may be the record address or the value of a specific key field or a combination of several key fields. The rest of the information which will be used only for output purpose is stored in the data fields of the records. Logically, the data structure can be depicted in Figure 2.1.

key Field 1	key Field 2	.....	key Field m	data Field 1	......	data Field n

Figure 2.1 A record of m key fields and n data fields.

The records in a file are uniquely-keyed if m=1 and multiply-keyed if m>1. A key is a generic-key if it appears in more than one record in the file; otherwise it is called a non-generic-key (Lefkovitz 1974). Of course, in a uniquely keyed file the key has to be non-generic in order to serve as the identifier. The structures of file organizations useful for files with uniquely-keyed records and for files with multiply-keyed records differ in many respects. A more detailed discussion of these structures is included in Section 2.5.

2.3 Basic Processing Requirements in Information Systems

Querying and updating are the two primary operations encountered in information systems. This is true in almost all existing and future systems. For example, in the Information Management System produced by IBM, the set of verbs GET UNIQUE, GET NEXT, INSERT, DELETE, REPLACE together with some other verbs of minor variation are incorporated in a procedural host language (IBM 1973). In the proposal made by the CODASYL Date Base Task Group (CODASYL 1971), the primary operators are FIND, GET, STORE, MODIFY, DELETE, etc.. Codd proposed for relational data system the following operators in his ALPHA language (Codd 1971) : GET, UPDATE, HOLD, PUT, DELETE, etc.. In the SMART retrieval system (Salton 1968), users query the system through natural language.

There are generally two types of queries. The first type of query retrieves a single record with its unique identifier. This is called primary key retrieval. The other type is secondary key retrieval that deals with files of multiply keyed records. A secondary key retrieval query specifies a retrieval criterion consisting of some constraint on a set of keys in the records. Updating operations involve addition or deletion of a key in a record, modification of a data field in a record, and addition or deletion of a record in a file.

2.4 Some Typical Information Systems

The two typical information systems to be discussed and compared in this section are attribute based record system and keyword based record system. They may also be discussed under the heading of data base management system and document retrieval system respectively. The distinctions and similarities of the two systems lie in the scopes of application, record requistion and formatting, query specification and retrieval criteria, etc.. As information systems become more integrated and generalized in the future, there is a good perspective of producing a unified system that incorporates features from both types of systems into a more generalized retrieval system.

2.4.1 Attribute Based Information Systems

A good but simplified example of an attribute based record system is the management and processing of personnel files in a small company. The data base contains an employee file EMP and a department file DEPT. File EMP is intended to describe each employee's pass number, name, department number, manager, position, telephone number and home address. File

DEPT contains data about department number, department name, department manager and location. Having decided on the data items to be included in the records of a file, the designer selects a fixed set of attributes that serves as a prototype for record construction. A record occurrence in the file is obtained by assigning a value to each of the selected attributes. Figure 2.2a illustrates examples for the EMP and DEPT files.

To put the above ideas into more formal terms, let each file have an associated set of attributes $= \{ A_1, A_2, \cdots, A_m, \bar{A}_1, \bar{A}_2, \cdots, \bar{A}_n \}$ where $A_1, A_2, \cdots, A_m$ are <u>key attributes</u> and $\bar{A}_1, \bar{A}_2, \cdots, \bar{A}_n$ are data attributes. Each key attribute A_i (or data attribute $\bar{A}_i$) takes on a value from its domain V_i (or $\bar{V}_i$) for any record occurrence in the file. A key field (see Figure 2.1) in such a record is then a key-attribute-value pair, $(A_i, V_{i,j})$, where $V_{i,j}$ is an element in domain V_i. Similarly a data field is a data-attribute-value-pair, $(\bar{A}_j, \bar{V}_{j,k})$, where $\bar{V}_{j,k}$ is an element in domain $\bar{V}_j$. An attribute should be designated as a key attribute if it will be used to specify retrieval criteria in the queries. If an attribute contains information which is only used for output purposes, then it should probably be considered as a data attribute. The only real significance in distinguishing a key attribute and a data attribute is that some physical retrieval paths will be built among the records and these retrieval paths generally depend on the values of the key

EMP file

attribute set = { PNO, NAME, DNO, MGR, POS, TEL, ADDR }

	PNO	NAME	DNO	MGR	POS	TEL	ADDR
r1	15	SMITH	1	15	Manager	6-4117	120 State St.
r2	20	DOE	1	15	Programmer	6-4118	76 First Ave.
r3	21	BAKER	2	21	Manager	6-4120	7 State St.
r4	22	TODD	1	15	Programmer	6-4119	21 Court St.
r5	27	ASTEN	2	21	Programmer	6-4121	32 Tioga St.

DEPT file

attribute set = { DNO, NAME, MGR, LOC }

	DNO	NAME	MGR	LOC
r1	1	Business	15	MALOT
r2	2	Purchase	34	DAY
r3	3	Programming	21	UPSON

Figure 2.2a Examples of attribute based records.

Query : Select from file EMP all employees who are in Department 1.

```
SELECT      NAME
FROM        EMP
WHERE       DNO = 1 ;
```

Figure 2.2b A query for EMP file.

attributes. For the EMP file in the example of Figure 2.2a, PNO, NAME, DNO, and MGR are possible choices for key attributes while TEL and ADDR should probably be data attributes.

Files of this nature mostly contain multiply keyed records and the keys are generally generic keys. Queries are normally expressed as a Boolean function together with a set of output key attributes and/or data attributes. The Boolean function is composed of keys and some logical operators. Those records and only those records that satisfy the Boolean retrieval constraint will be retrieved and the contents of the output attributes are presented to the user. Figure 2.2b shows a typical expression for such a query.

Date (1975) gives an excellent review of three most popular design architectures for attribute-based information systems. The reader may refer to the book for a more detailed description of this type of system.

2.4.2 Keyword Based Information Systems

The SMART retrieval system (Salton 1968, 1971, 1975) is a good example of a keyword based information system. Such a system is designed to furnish pertinent references in answer to specified search requests. In the literature search environment of libraries, the entities to be retrieved are books or articles published in journals and these documents are available in natural language texts. The search requests can be a set of user supplied keywords or, in some more sophisticated systems like SMART, they may simply consist of natural language sentences. The immediate problems to be tackled include the ways of content indication and of computer record representation for texts. These problem in keyword based systems are different from the corresponding ones that arise in attribute based systems. First of all, it is not easy to select a fixed set of attributes that can properly indicate the contents of texts since the texts may describe ideas or concepts which are not always instances of concrete objects like employees or departments or autoparts etc.. Secondly, the document or query texts are not always semantically precise because human intelligence is involved in judging the meaning of a query or a document as well as the relevance of one with respect to the other. In fact, it is very likely that a retrieved document may prove to be highly relevant to one user but is only marginally so for another even though both users submit exactly the same query. Therefore, one cannot always tell with a hundred percent confidence that a document is exactly (or is exactly not) relevant to a user. The retrieval system then has to "guess" intelligently the degree of interest with which a user wants to see a particular document in order to retrieve those documents which are most likely to be useful. Because of this uncertainty in query and/or document meaning and in the relevance between them, the retrieval procedure is a "fuzzy" one.

The fuzziness is generally expressed as a similarity measure (Salton 1968) which estimates the degree of relevance between a query and a document. Record representation and retrieval are much more difficult problems in such systems than in attribute based systems.

Records in keyword-based systems are generally constructed from a set of attributes together with a group of keywords. The set of key attributes for library documents can be {Author, Date of publication, Journal of publication Language} and the set of data attributes can be {Citation, number of copies in library}. Such a set of key attributes clearly does not indicate the content of the text appropriately. Hence a set of keys must be selected to augment the content indication. An example of a document record is shown in Figure 2.3.

Queries can be formulated in several ways. One possibility is a constraint on the key attributes, for example, "retrieve all papers written by John Smith published in BACM before July 1974". Effectively, this is equivalent to the Boolean expression (Author = John Smith) $\wedge$ (Date of publication < July 1974) $\wedge$ (Journal = BACM). A second kind of query specifies a set of keys indicating the contents of the documents in which the user is interested. In such a situation, records containing some or all of these keywords should be considered and evaluated for their possible degree of relevance. A commonly used similarity measure is

Document : A paper written by John Smith on selecting optimal paging size using computer simulation technique. The paper is published in journal BACM on October 1971.

Document record representation :

{attribute-value-pairs} = {(Author, John Smith),
(Date of publication, October 1971),
(Journal of publication, BACM), (Language, English)}

and

{Keyword} = {optimal, page, size, computer, simulation}

Figure 2.3 Record representation of a sample document.

the cosine function (Salton 1968) which computes the number of matched keywords between a query and a document divided by a length normalization factor. This produces a quantity between zero and one. The cosine measure is equal to one if the set of keywords in the document is exactly the set of keywords in the query. When a document has no keyword in common with the query, the cosine measure has value zero. It is hoped that the larger the cosine measure the more likely it is that the document will satisfy the user's need.. The last kind of query consists in combining the two previous ones. This kind of query retrieves documents by keywords together with some restrictions on the values of key attributes.

The retrieval constraints on key attributes can be written by the user as a Boolean expression. When the constraint is written in natural language, Weiss (Weiss 1970) has shown that it can be effectively translated into a Boolean form automatically.

The keywords in a query can either be picked manually or extracted automatically from text. When the user picks keywords manually, he can designate some logical relationships (AND, OR, NOT, etc.) between them if he chooses to. Otherwise the keywords are merely considered as being connected by the logical OR relation. If queries are submmitted as natural language sentences, then some indexing method should be adopted (Salton 1974) to automatically extract keywords from the query texts. The indexing methods generally consist of procedures which select a set of keywords and determine the relationships between them. Again, only OR relations exist between keywords in the most simple-minded indexing methods. Some recent indexing methods include a thesaurus and a phrase generation method as well as logical AND and OR relations between keywords (Salton 1974, Salton 1975).

Hence, in general, whether a query is originally submitted as a Boolean expression or not, it can always be considered as being in Boolean form with or without any application of indexing methods.

In the following survey of file organizations, queries for both attribute based information systems or keyword based information systems are considered to be in Boolean form. This allows one to treat file organizations for both systems in a more uniform manner. Although some search mechanisms are different for the two systems, most file organizations are applicable in either case.

2.5 Computer System for Information Processing

In contrast to scientific applications where complicated and prolonged arithmetic and logical operations are performed on relatively small amounts of data, information processing systems typically handle very large volumes of data with the types of computations that are comparatively trivial. Under the present state of computer technology, it is not possible to store in the main memory even a small portion of the data base used in an information system. The data base is therefore placed in backup storage devices (disks, drums, tapes, etc.) and only small pertinent parts are brought to main memory during the processing of a query. However, the speed of data transfer between backup storage and main memory is about three orders of magnitude slower than the computational speed in the central processing unit. It is therefore important to reduce the amount of data transmitted as much as possible, and this is exactly the motivation for designing sophisticated file organizations. Hence, it is convenient to inspect data transmission actions arising in a typical computer system before reviewing various file organizations.

Figure 2.4 depicts a multiprogramming computer system which is most popularly encountered in information processing environments (Lefkovitz 1974). Normally, a large number of controllers can be attached to an I/O processor. In this

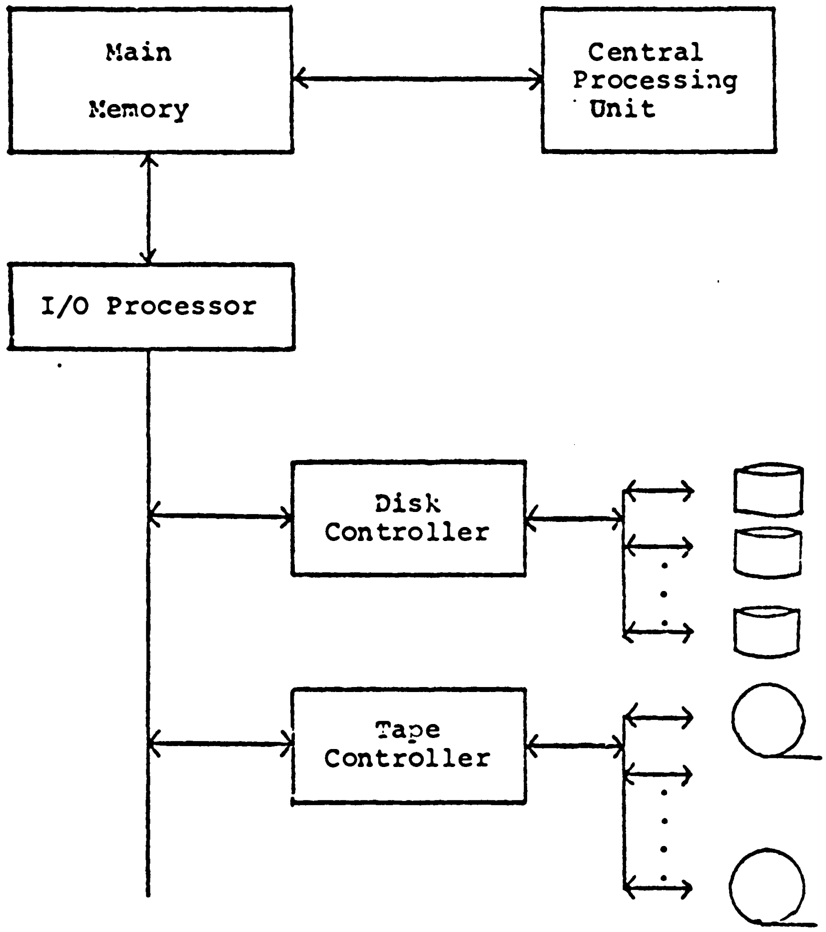


Figure 2.4 Computer System Configuration

diagram, only one disk controller and one tape controller are shown for the purpose of simplicity. A controller is connected to several storage devices of the same type. The I/O processor receives an I/O command from the CPU and passes it on to the appropriate controller. In the case of a disk controller, the read head of the requested disk is made to move to the selected cylinder containing the requested data. The time needed for such a disk head movement is called seek time. After the seek operation, data transmission can not begin until the controller becomes available to this particular disk. Once the controller is available, data transmission will be started after the disk rotates to the point where the beginning of the requested block is located beneath the read head. Such rotational delay, generally termed latency time, is, on average, half of the disk rotational time. Following the latency time is the transmission time needed to transfer data from disk to controller, which is proportional to the length of data requested. Having buffered data from disk, the controller transfers the whole block to the main memory via an I/O processor. It should be noted that the data transfer rate between main memory and controller is much faster than that between the controller and the disk. Normally, the disk controller can set the various heads into motion simultaneously and hence the seeking activity is overlapped in a single controller. On the other hand, the transmission path between the controller and the main memory is not shared. Because of this, it is sensible to assume that the amount of time needed to access a block of data is the sum of the average latency time and data transmission time. This assumption is used throughout this thesis for timing purposes.

The access considerations for drums are the same except that drums always have zero seek time. Access actions for tapes are not of concern here since this thesis concerns itself solely with on-line information processing which would never use tapes.

2.6 A Survey of File Organizations

2.6.1 A Generalized Diagram for File Organization and Search Algorithm

Five general classes of file organizations are reviewed in this section. Discussion is centered on the ability of each file organization to support on-line and/or off-line processing and primary and/or secondary key retrieval as well as on the processing and storage costs of the system.

Figure 2.5 presents a generalized schematic diagram of most file organizations and of the associated search procedures. It will be seen later that particular realizations of components in the diagram produce different classes of file structures. The upper part of the diagram shows the search procedure and the lower part is the data base. Search actions can be illustrated by the following steps :

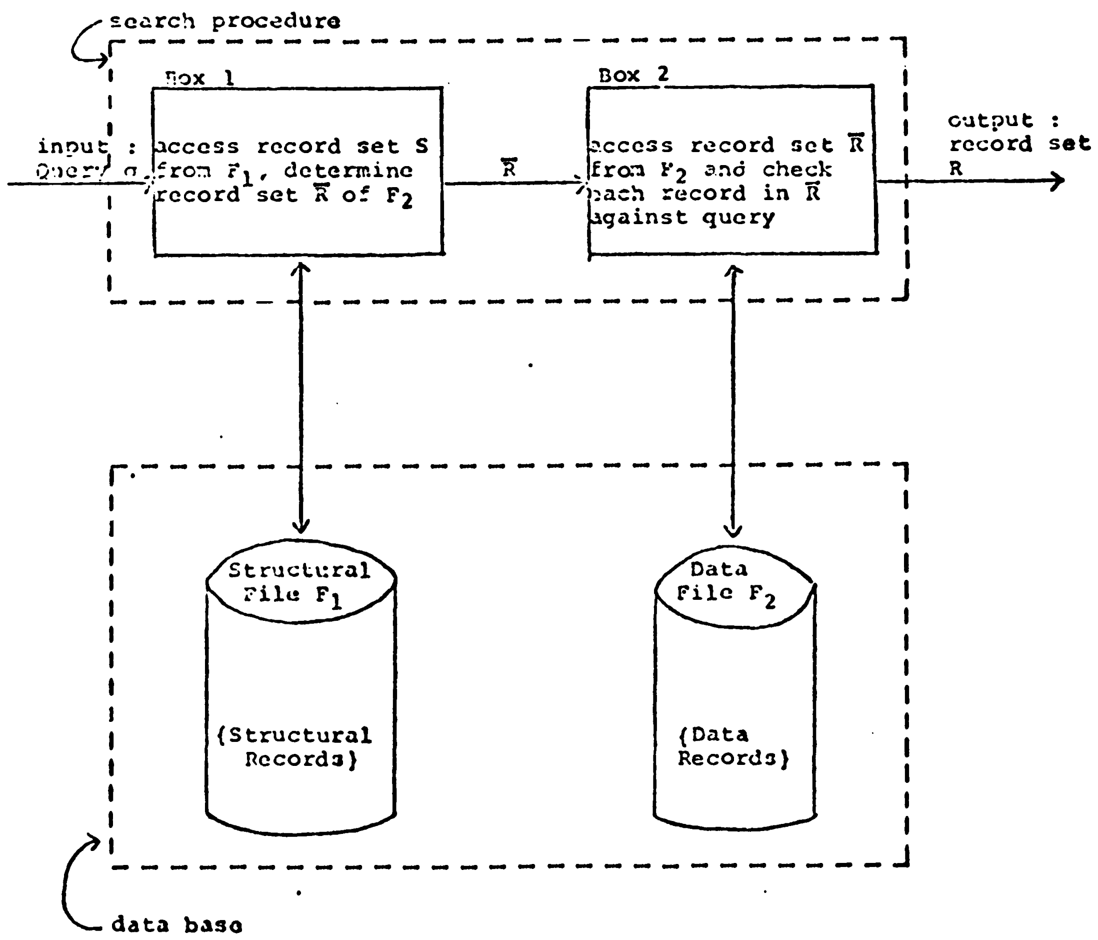


Figure 2.5 A generalized diagram for file organization and search process

Algorithm G (a generalized file retrieval algorithm)

Step 1 [input query] A retrieval criterion is specified in query q.

F_2 is the file that contains data records to be retrieved. In most applications, it is undesirable to search all records in F_2 for each query. Since a retrieval criterion is included in the query, it would be beneficial to use this criterion in order to restrict the scope of search to a smaller subset, $\bar{R}$, of F_2. In most file organizations, this is achieved by either of the following two methods. The first is to build certain physical access paths in the file using some keys in data records. A second method is to assign addresses of data records in F_2 by a computational procedure (hashing, say). In the case where physical access paths are constructed, an additional file F_1 is required to store the necessary structural information. During retrieval, the system has to access a set of structural records, S, from F_1 to determine a record set $\bar{R}$ of F_2, where $\bar{R}$ is the set of records that contains the the answer set R. If, on the other hand, the addresses of records in F_2 are allocated by some computational procedure, the computations should be performed on queries to determine the addresses of records in $\bar{R}$.

The address space of records in $\overline{R}$ is usually confined to a few small local regions in F_2. In general the costs associated with Box 1 in Figure 2.5 are the cost to access records S and/or the cost of doing the necessary computations. Therefore, a reasonable file organization is one which accesses a set S with small or zero cardinality and does little computation in the CPU while at the same time procuding a set $\overline{R}$ which is a small superset of the answer set R.

The above paragraph explains the contents of files F_1 and F_2 and the processing involved in box 1 of the diagram. Step 2 summarizes these considerations :

Step 2 [determine $\overline{R}$] If a structural file F_1, exists, then access structural record set S, process records in S and determine $\overline{R}$. For file organizations where F_1 does not exist, apply some computational procedure to the query q and determine $\overline{R}$.

Assume that R is the set of records which satisfy the query criterion. For most file organizations, it is unlikely that Step 2 can determine $\overline{R}$ so precisely that $\overline{R}$ is exactly the same as R. Hence records in $\overline{R}$ must actually be accessed from F_2 and checked against the query criterion. This is the processing included in Box 2 of the diagram. The cost at this step depends on the size of set $\overline{R}$ as well as on the actual locations of the records $\overline{R}$ in the file. If the size of $\overline{R}$ is reduced or the addresses of records in $\overline{R}$ are localized in the storage space, the access cost of records in $\overline{R}$ is also reduced. This part of the processing can be described as follows.

Step 3 [Check $\overline{R}$ and output R] Access data records in set $\overline{R}$ from F_2. Present to the user each data record that satisfies the retrieval criterion.

Under the general framework developed in this section, the characteristics of sequential organization, indexed sequential organization, list structure organization, clustered organization and scatter storage organization are discussed in the next few sections.

2.6.2 Sequential File Organization

The sequential file organization is conceptually simplest. The records are not sorted in any preselected order, no subfiles or file partitions are defined, and no directories exist which provide access to any subportion of the file. In terms of the diagram in Figure 2.5, F_2, the data file, is the sequential file and F_1, the structural file, does not exist at all. Consequently, S, the subset of structural records, is null and $\overline{R}$, the subset of F_2 to be accessed and checked, is the entire F_2. With such a minimal setup, there is no better way to tackle the search problem than to serially scan every single reocrd in the file until the requested one is reached. More specifically, to retrieve a record of a given key, the average number of accessed records is (N + 1) / 2 for a successful search and N for an unsuccessful search where N is the number of records in file F_2.

Some improvement to the efficiency of a successful search can be achieved if a non-uniform access frequency pattern is associated with the data records. Knuth (1973) described a "move-to-front" heuristic which moves a record to the beginning of the sequential file whenever a record is requested. In the steady state, the records will be approximately aligned in decreasing accessing frequency order and the average successful search time will henced be reduced. If the underlying frequency distribution obeys Zipf's law (Zipf 1949), the average number of records accessed in a successful search is lowered from (N + 1) / 2 to approximately $2N / \log_2 N$. Rivest (1974) proposed another "transposition" method which exchanges the requested record with its immediate predecessor and reduces the search time further than the "move-to-front" approach.

With or without the self-organizing adjustment, the excessive amount of searching required of a sequential file rules out its use for real time applications. But in some off-line environments where a large number of queries are processed as a batch, a sequential file may outperform many more sophisticated file organizations. Furthermore, a sequential file uses the least possible storage space, since no links, pointers, or other structural data are incorporated into the file. Search and file maintenance procedures are also relatively trivial tasks.

2.6.3 Indexed Sequential Organization

The indexed sequential file organization uses directories to speed up the search in the data file. It can be implemented on any direct access devices (e.g. disk, drums, etc.). For the purpose of simplicity, movable disk packs are adopted for illustration in this section.

As pointed out in section 2.2, each record in a file is uniquely designated by an identifier. For an indexed sequential organization, records in the data file F_2 are stored in the order of the identifiers either by physical proximity or by using explicit pointers. File F_1 contains a collection of track indexes and a cylinder index if file F_2 occupies more than one cylinder. There is one track index for each cylinder of the file and the track index is stored at the beginning portion of the first track of that cylinder. The i^{th} entry in the cylinder index contains the identifier of the last* record in the i^{th} cylinder. Similarly, the j^{th} entry of a track index contains the identifier of the last record in the j^{th} track.

To retrieve a record by its identifier, one first consults the cylinder index to ascertain the address of a cylinder in which the record is possibly stored. Then the track index of the specified cylinder is looked up to locate a certain track which will be scanned linearly in an effort to find the record. This process is illustrated in Figure 2.6.

The initial loading of an indexed sequential file generally assigns physically consecutive addresses to records of consecutive identifiers. When a new record is to be added to a particular track as determined by its identifier value, the "home track" may have already run out of available space. This implies that an overflow area has to be reserved to accommodate overflow records. Furthermore, the overflow records must be linked to the records in the home track in order to preserve the logical record sequence in the file.

The cylinder index is usually small and can be stored in the main memory during file processing. Thus, the total number of disk accesses required to retrieve a record with a given identifier consisting of the one access to a track

---

* If records are stored in increasing (decreasing) order of the identifiers, "last" means the record with the highest (lowest) key.

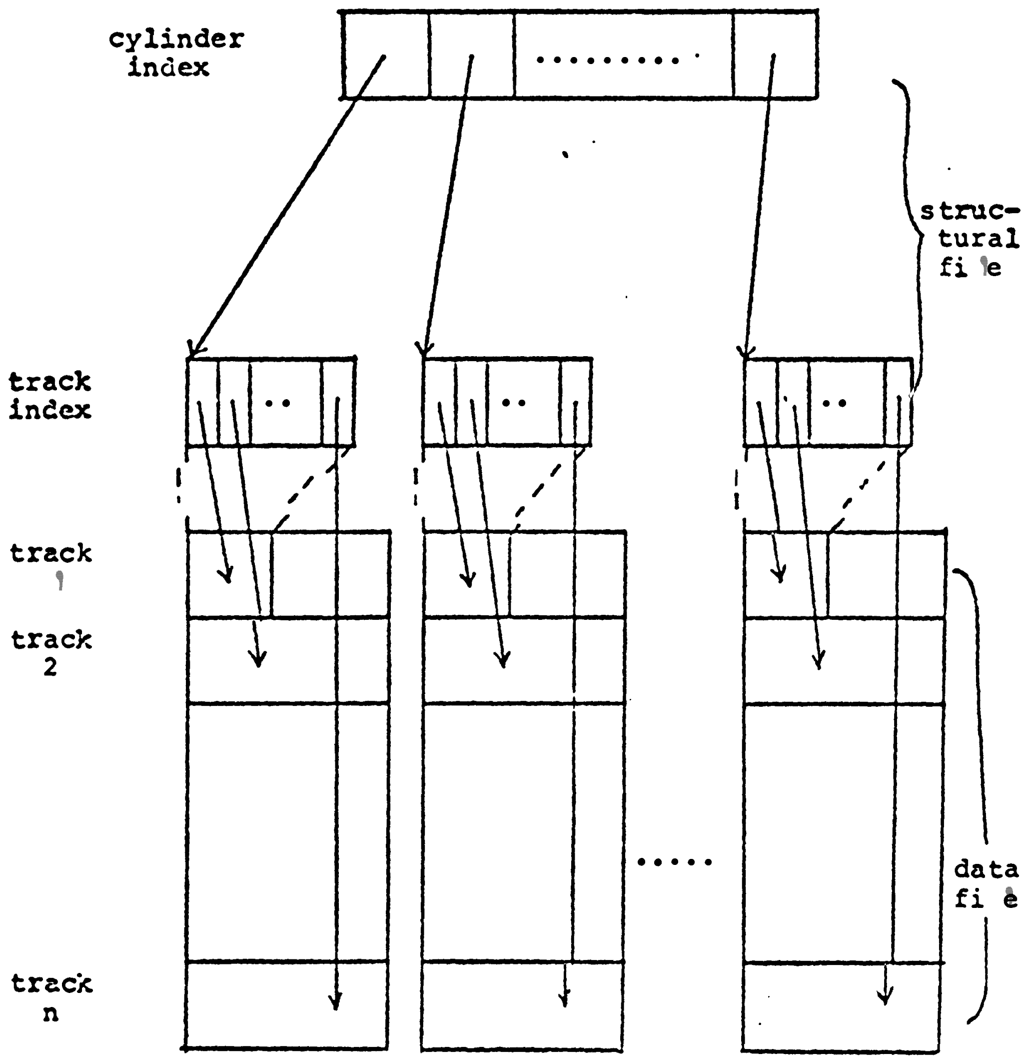


Figure 2.6 Indexed Sequential File Organization

index plus the number of accesses needed to scan records on a selected track including also the possible overflow records associated with this track. This greatly reduces the required number of accesses needed to retrieve a record compared with the accesses needed in a sequential file organization. In fact, the indexed sequential organization is widely adopted for real time information retrieval.

When a considerable number of records are deleted from an indexed sequential file, the storage space typically contains a good deal of wasted empty slots. Also, the insertion of new records tends to render the overflow chains too long and deteriorates the search performance. Hence a periodic reorganization of indexed sequential files is essential for files with high activity of updating.

It should be noted that an indexed sequential organization is suitable for primary key retrieval where only a single identifier is used for retrieval. In the case of secondary key retrieval, the indexed sequential organization must be used with additional list structures in order to retrieve the required data records. This will become clear in section 2.6.5.

2.6.4 Scatter Storage Organization

In the scatter storage file organization, the structural file F is eliminated from the diagram of Figure 2.5. A computational procedure is adopted instead to determine the record addresses. Given a query q, algorithm G applies this computational procedure to q so that record accesses are restricted to small portions of the data file F_2.

More precisely, the scatter storage organization partitions the available storage space into a group of buckets of the same size. One or more records can be allocated to each bucket depending on the bucket and record sizes. A mathematical transformation, known as a key-to-address transformation, or hashing function, is then used to map from the keys associated with each record into a bucket address in the storage space. A collision is said to occur when a new record hashes to a bucket which is already full. In case of collision, an overflow technique is necessary to assign the new record to some other bucket. The important problems in the scatter storage organization are therefore

1) the choice of a good hashing function which minimizes the number of collisions for a set of keys in a file, and
2) the design of an overflow technique which minimizes the lengths of the overflow chains.

Morris (1968) and Knuth (1973) have analyzed both problems in detail. Extensive experimental results can be found in Lum, Yuen and Dodd (1971).

The scatter storage organization is applicable to both primary key retrieval and secondary key retrieval. For primary key retrieval the task is relatively easier since only one key in the record or query is used for the key-to-address transformation. For secondary key retrieval, a variable number of keys in the query or record must be used in the transformation. Files and Huskey (1969) suggested a so called superimposed coding which consisted in mapping each key into an n-digit number. The n-digit numbers corresponding to the individual keys in a record are then transformed into a single composite code and this code is in turn mapped into a storage address. Gustafson (1969) proposed a hashing technique for the special case where each record had the same number of keys, say k keys. Each record is converted to an n-bit mask where exactly k bits are set to one and n-k bits are set to zero. Each key in a record is hashed to a bit position between one and n and the corresponding bit in the mask is set to one. When some keys in a record hash to the same bit position, additional bit positions are randomly generated so that exactly k bits are set to one. The resulting bit string can then be used to calculate a bucket address.

The number of buckets accessed in a successful or unsuccessful search depends solely on the loading factor of the storage space. When the scatter storage is used for primary key retrieval, for a loading factor up to 0.8 and a reasonable bucket size, the average number of buckets accessed is usually less than two for pratically all the hashing functions and overflow handling techniques tested by Lum, Yuen, and Dodd

(1971). The scatter storage technique is therefore a strong candidate for systems which require fast real time processing.

Scatter storage is not without disadvantages. In the worst case, the average number of backets accessed can be very large. Some storage space is wasted because the loading factor must be relatively small in order to maintain fast retrieval. When the storage space is mostly filled, there is no easy way to expand the space without having to rehash the whole file. Also, no specific record ordering exists in the file.

2.6.5 List File Organization

In general, list file organizations associate each key with a set of records are described by this key. There are several variations of the list file organization, each corresponding to a specific implementation of the above association. Inverted list organizations and multilist organizations are two notable examples among them.

The structure of an inverted list organization is shown in Figure 2.7. The data base can be partitioned into three components : a directory file, a list file and a data file. The data file contains the data records as usual and can be identified as file F_2 in the generalized diagram of Figure 2.5. The list file and the directory file jointly constitute the structural file F_1. There exists in the list file exactly

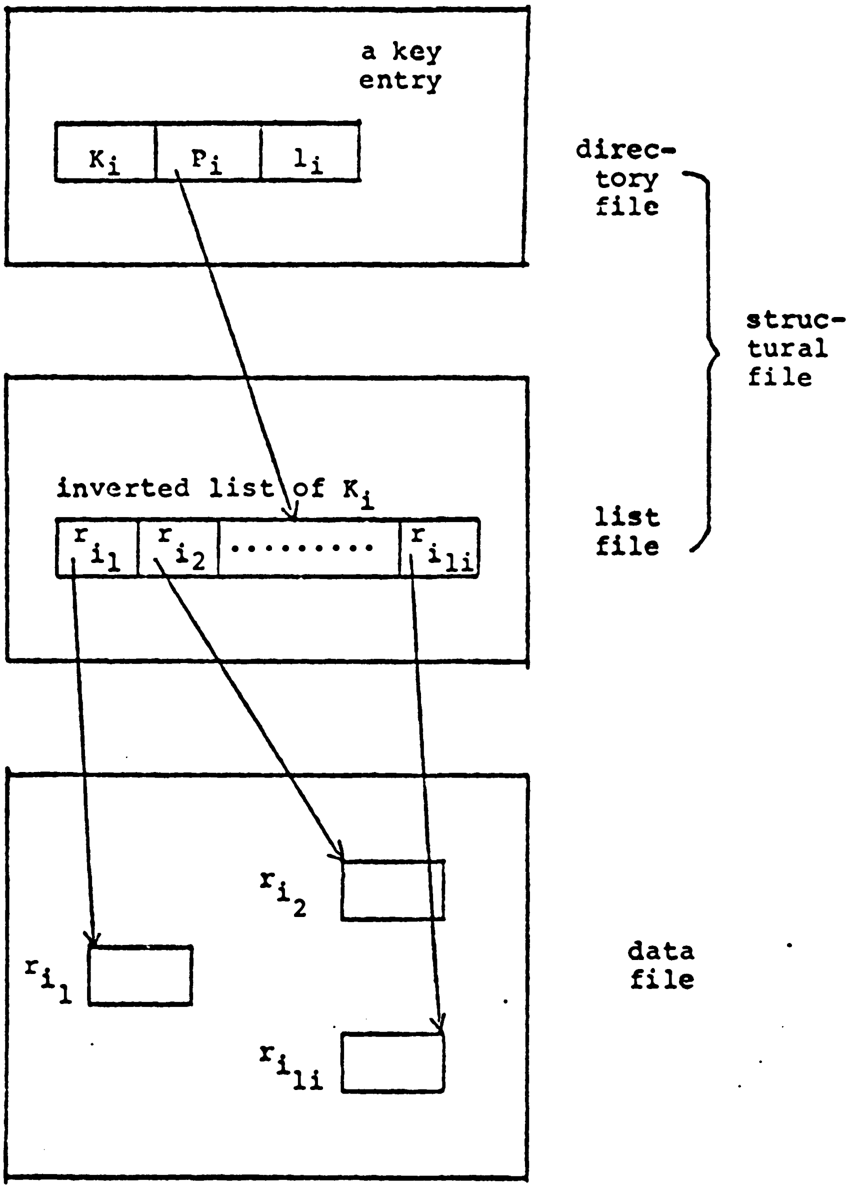


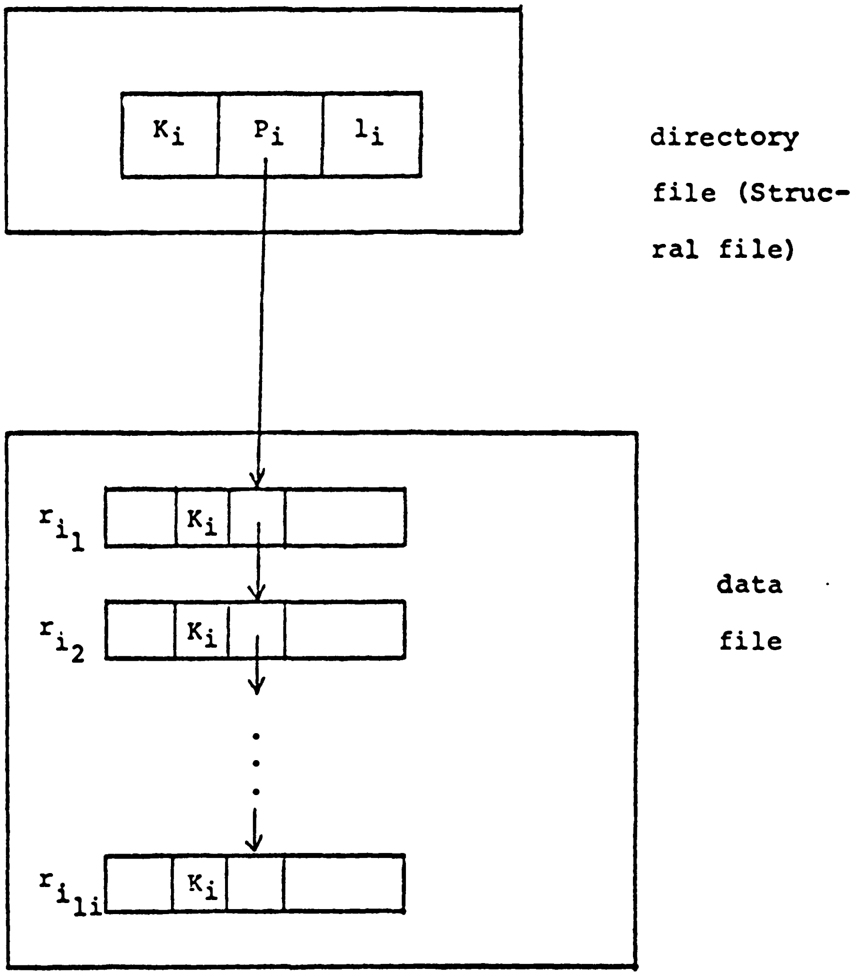
Figure 2.7 Inverted List Organization

one "inverted list" for every key in the file, and this inverted list includes one identifier for each data record which contains the corresponding key. The directory contains one entry for every key used in the file. The information stored in a key entry consists of the key K_i itself, a pointer P_i which points to the inverted list of this key, and a length l_i which indicates the total number of records in which K_i appear. This construction explains how the set of data records associated with any given key is retrieved.

The multilist organization shown in Figure 2.8 is obtained by merging the list file and the data file of an inverted list organization. Each key field in the data record is augmented by a pointer subfield which points to a next data record that contains this same key. The P_i field in a directory entry points to a data record which has key K_i, the other two fields have the same significance as before.

A few other list organizations can be obtained by modifying either the inverted lists or the multilists (Lefkovitz 1969). Chapter three of this thesis uses inverted lists and multilists to construct a new class of list organizations.

List file organizations are designed primarily for secondary key retrieval. If the query is a Boolean expression of more than one key, the inverted list organization retrieves and manipulates all the associated lists in order to determine the set of data records that are exactly pertinent to this



<u>Figure 2.8</u> Multilist Organization

query. The multilist organization accesses records in some selected lists and may possibly access certain undesirable records. Detailed search algorithms will be discussed in the next chapter. In general, when the lists are long, an inverted list organization tends to be more efficient than a multilist organization and vice versa.

List file organizations are much faster than sequential files. But this improvement is obtained at the cost of additional storage and maintenance due to the existence of structural files. In the case of an inverted list organization, any specific record identifier is stored in the list file as many times as the number of keys which exist in the record. The size of the list file is thus about the same as the size of the data file. Furthermore, a directory file is also needed. Hence the required storage space is much larger than for a plain sequential file. A multilist organization uses exactly the same amount of storage as inverted list organization.

The repetition of information storage in structural and data files naturally leads to duplicated file maintenance tasks. The directory file can be implemented as an indexed sequential organization, a scatter storage organization, or any other feasible structure. When keys are added to or deleted from the directory, the required work for an indexed sequential directory or a scatter storage directory is the same as that treated earlier in sections 2.6.3 and 2.6.4.

The inverted lists and multilists also must be modified whenever data records are inserted or deleted or modified. The file updating procedures for list organizations will receive more careful analysis in the next chapter.

2.6.6 Clustered File Organization

The clustered file organization partitions the data file into groups of related records by some classification algorithm. Ideally, the classification algorithm should group records in such a way that items in the same group are always jointly requested by queries. Unfortunately, it is not possible to achieve this ideal situation for all queries. Hence, a realistic goal of classification is to quarantee that any query can locate all requested records in a small number of groups. Normally, a similarity function is used to estimate the closeness between any two items. The similarity function should be designed in such a way that a high similarity measure between a pair of records indicated that the keys in the two reocrds are closely related and that the two records will be likely to be jointly relevant to the queries. The classification algorithm must then assign highly similar records to the same group and dissimilar records to separate groups.

Each group can in turn be identified by a group profile, or centroid, which is representative of its member records. A query must first search through the centroids, and then access member records for highly correlated groups only. The above process can be generalized by constructing as many levels of centroids as necessary. The resulting clustered organization looks like the multiple level tree exhibited in Figure 2.9. Some desirable properties of clustered trees are studied in chapter six.

Comparing the generalized file organization diagram of Figure 2.5 with the clustered tree in Figure 2.9, the data records which are the leaves of the tree constitute the data file F_2, and the rest of the clustered tree represents the structural file F_1, which is essentially a directory in this case.

As noted before, clustered organizations make it possible to probe only a small percentage of the groups for each query. Moreover, another advantage is the flexibility in search scope adjustment. If a query requires the retrieval of a large number of records, a broader portion of the tree is searched. On the other hand, only a small portion of the tree needs to be probed if only a few records are requested by the same query.

The classification of records is usually a difficult and time-consuming task because of the large number of records in the data file. Williamson (1974) proposed a clustering algorithm with a worst case as well as average case running time of O(NlogN) where N is the total number of data records.

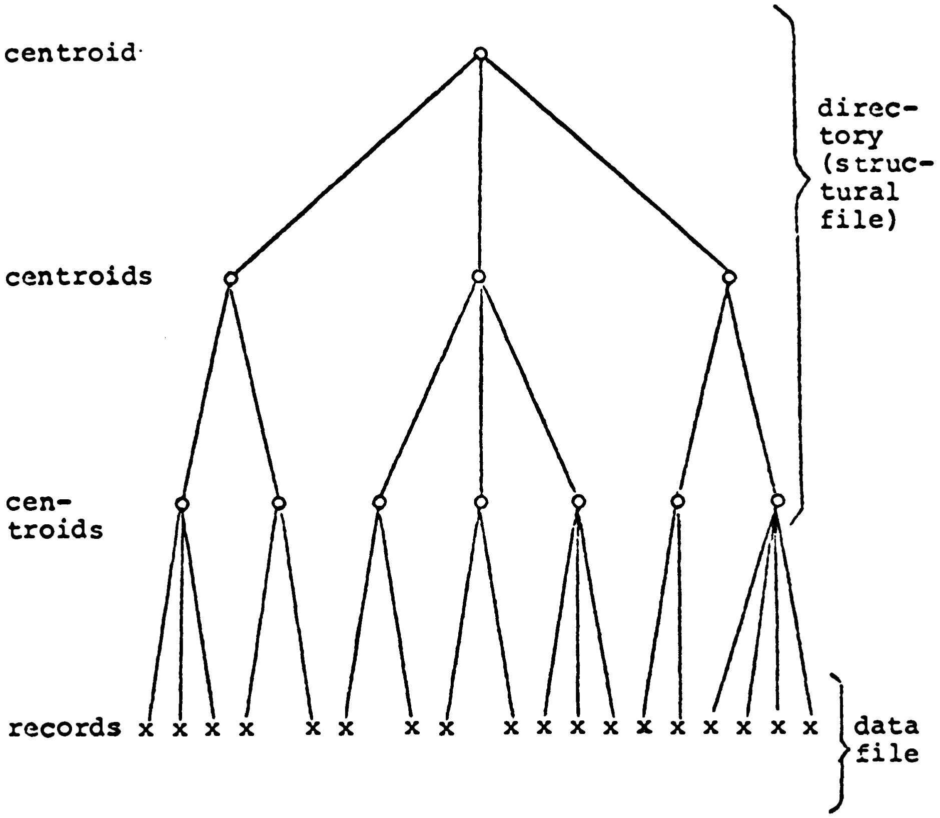


Figure 2.9 Tree Structure of a Clustered Organization

Williamson's method runs considerably faster than most other algorithms. But an O(NlogN) process may still be costly since data files may typically be very large in many information systems. It is desirable to design even more inexpensive methods. Many other interesting clustering algorithms can be found in Bonner (1964), Litofsky (1969), Dattola (1973), Yu (1973) etc..

2.6.7 Summary

Five principal file organizations for handling large volumes of data on backup storage devices were surveyed in this section. Each of them is actually a special realization of the generalized file organization schema presented in section 2.6.1. The sequential file organization is really applicable to batch processing only. The indexed sequential organization alone does not facilitate secondary key retrievals although it is very useful for on-line primary key retrieval. Scatter storage organizations can be utilized for both primary and secondary key retrievals; they also provide fast response time as required in real time file processing environments. List file organizations and clustered file organization are designed mainly for on-line processing of secondary key retrievals. Salton (1975) also discusses these organizations and tabulates their individual characteristics as well as comparative results.

Chapter 3 A Structure and Search Algorithm for a Class of List File Organizations

The structure of inverted list and multilist organizations was discussed in section 2.6.5. In this chapter, querying and updating costs for both organizations are analyzed in more detail in sections 3.1 and 3.2. An algorithm is proposed in section 3.1.2 to avoid redundant record accesses involving unsorted multilists. Section 3.3 presents a new class of list file organizations, the hybrid list files. Elements in this class are determined by assigning values to a list length parameter l_{th}. Both the inverted list organization and the multilist organization are actually special cases of this new class. Section 3.4 presents an analytic result concerning the selection of an optimal file element from this class by assuming some simplified system characteristics. For more complicated system environments, simulation techniques are used in the last section to help select optimal elements from the class of hybrid file organizations.

3.1 Querying Costs for Inverted List and Multilist Organizations

In chapter two it was concluded that queries in both a attribute based information systems and keyword based information systems could always be formulated as Boolean expressions. It is a well known fact that any Boolean formula can be expressed in disjunctive normal form (DNF) as a sum of products of literals. Literals in the context of Boolean queries are keys and/or negations of keys. From now on, queries are assumed to be in DNF's with the following notation :

$$Q = \bigvee_{i=1}^{m} (K_{i,1} \wedge K_{i,2} \wedge \cdots\cdots\cdots \wedge K_{i,n_i})$$

i.e. a sum of m products, the i^{th} product being composed of n_i keys and/or negations of keys. Designating the i^{th} product of Q by P_i, Q can be rewritten as $Q = \bigvee_{i=1}^{m} P_i$. Throughout the discussion, it will be assumed that, in each product P_i, at least one of the literals is a non-negated key. This is almost always true in practice. Since list organizations commonly construct lists for non-negated keys only, it would be very inefficient to process query products consisting completely of negated keys.

Let $l_{i,j}$ be the number of records in which key $K_{i,j}$ appears and, without loss of generality, assume that $\ell_{i,1} \leq \ell_{i,2} \leq \cdots\cdot \leq \ell_{i,n_i}$ is true. Also let ℓ_i and ℓ be the number of records satisfying P_i and Q respectively. The definitions of symbols used in this chapter are collated in Table 3.1.

Q	: $\bigvee_{i=1}^{m} (K_{i,1} \vee K_{i,2} \vee \cdots \vee K_{i,n_i}) = \bigvee_{i=1}^{m} P_i$, query expressed in disjunctive normal form.
$\ell_{i,j}$	: number of data records in which key $K_{i,j}$ appears.
ℓ_i	: number of data records in which keys $K_{i,1}$, $K_{i,2}$, ..., K_{i,n_i} appear simultaneously.
ℓ	: number of data records which satisfy query Q.
w	: number of data records which satisfy $K_{1,1} \vee K_{2,1} \vee .. \vee K_{m,1}$.
$L_{i,j}$	: the inverted list corresponding to key $K_{i,j}$.
L_i	: the set of data records in which keys $K_{i,1}$, $K_{i,2}$, ..., K_{i,n_i} appear simultaneously.
L	: the set of data records which satisfy query Q.
T_L	: average latency time of the device holding list file and data file.
T_t	: time needed to transmit one byte of data from the device holding list file and data file.
k	: average number of keys in a data record.
A	: number of bytes needed to represent a record identifier in the device holding list file and data file.
R	: average number of bytes per data record.
$\overline{T}_i$	: average amount of time needed to perform a set intersection operation in CPU.
$\overline{T}_u$	: average amount of time needed to perform a set union operation in CPU.

Table 3.1 Symbols used in Chapter 3.

C_{decode}	: average cost of decoding one key from the directory file.
C_{ch}	: cost of one unit of channel time.
C_{CPU}	: cost of one unit of CPU time.
C_K	: cost to write a key entry to the device holding directory file.
QC_I	: cost to process a query of the form $Q = \bigvee_{i=1}^{m} P_i$ in an inverted list organization.
QC_{IS}	: cost to process a query of the form $Q = K_{i,1} \wedge K_{i,2} \wedge \cdots \wedge K_{i,n_i}$ in an inverted list organization.
QC_M	: similar to QC_I but for multilist organization.
QC_{MS}	: similar to QC_{IS} but for multilist organization.
QC_H	: similar to QC_I but for hybrid list organization.
QC_{HS}	: similar to QC_{IS} but for hybrid list organization.
UC_I	: average cost of performing one file maintenance operation in inverted list organization.
UC_M	: same as UC_I but for multilist organization.
UC_H	: same as UC_I but for hybrid list organization.
X	: number of data records in which an arbitrary key appears.
$P_i (i=1,2,3,4,5)$	: proportion of type i update operations.
P_I	: proportion of inverted lists in a hybrid organization.

Table 3.1 (Continued)

P_M : proportion of multilists in a hybrid organization.

P_Q : proportion of transactions which are queries.

P_U : proportion of transactions which are updates.

<u>Table 3.1</u> (Continued)

3.1.1 Querying Cost for Inverted List Organization

Given query $Q = \bigvee_{i=1}^{m} (K_{i,1} \wedge K_{i,2} \wedge \cdots \wedge K_{i,n_i}) = \bigvee_{i=1}^{m} P_i$, let $L_{i,j}$ be the inverted list for key $K_{i,j}$ stored in the list file (Figure 2.7), and let L_i and L be the sets of records satisfying P_i and Q respectively. In other words the length of inverted list $L_{i,j}$ is $\ell_{i,j}$ and the sizes of sets L_i and L are ℓ_i and ℓ respectively.

The following procedure accesses inverted lists corresponding to keys in each product P_i and performs a list intersection after each list is accessed. When all the lists corresponding to keys in one product have been intersected, a set union is performed.

Algorithm I (Querying Procedure in Inverted List Organization)

Step 1 : $L \leftarrow$ empty set.

Step 2 : (list accessing and list intersection for each of the m products $P_1, P_2, \ldots, P_m$)
Repeat steps 3,4,5 for $i = 1, 2, \ldots, m$.

Step 3 : (access first list in product $P_i = K_{i,1} \wedge K_{i,2} \wedge \cdots \wedge K_{i,n_i}$) Decode key $K_{i,1}$ from directory file (see Figure 2.7), access list $L_{i,1}$, $L_i \leftarrow L_{i,1}$.

Step 4 : (access the remaining n_i-1 lists, after each access, perform one list intersection)
Repeat the following for $j = 2, 3, \ldots, n_i$:

Decode key $K_{i,j}$, access list $L_{i,j}$,

$L_i \leftarrow L_i \cap L_{i,j}$.

Step 5 : $L \leftarrow L \cup L_i$.

Step 6 : Access data records in set L. END.

The tasks performed in Algorithm I include key decoding, list accessing, set intersection, set union and data record accessing. The cost of Algorithm I can be easily formulated using the following variables :

T_L = average latency time of the device holding list file and data file,

T_t = time needed to transfer one byte of data from the device holding the list and data files,

A = number of bytes needed to represent a record address for the device holding the list and data files,

R = average number of bytes per data record,

$\overline{T}_i$ = average amount of time needed to perform a set intersection in CPU,

$\overline{T}_u$ = average amount of time needed to perform a set union in CPU,

C_{decode} = average cost of decoding one key from the directory file,

C_{ch} = cost of one unit of channel time,

C_{CPU} = cost of one unit of CPU time.

The querying cost using the inverted list organization is then :

$$QC = (\sum_{i=1}^{m} n_i)\, C_{decode}$$

$$+ \sum_{i=1}^{m} (n_i\, T_L + \sum_{j=1}^{n_i} \ell_{i,j}\, A\, T_t)\, C_{ch}$$

$$+ (\sum_{i=1}^{m} (n_i - 1)\, \overline{T}_i + (m - 1)\, \overline{T}_u)\, C_{CPU}$$

$$+ \ell\, (T_L + R\, T_t)\, C_{ch}$$

The first term in QC_I represents the cost to decode all keys used in query Q. The second term corresponds to the accesses of inverted lists from the list file. As pointed out in section 2.5, the time needed for each access is the sum of average latency time and data transmission time . The cost of list intersections and list unions is included in the third term, and the last term accounts for the time to access the ℓ data records which satisfy query criterion Q.

A precise formulation of C_{decode} depends on the organization adopted for the directory file. It can be an indexed sequential organization or a scatter storage organization. Because record identifiers in inverted lists are always sorted, the amount of time to perform a set union operation is actually a function of the sizes of the two sets involved; the same is true for the set intersection operation. For the time being, it suffices to use the average values, C_{decode}, $\overline{T}_u$, and $\overline{T}_i$, for the above three quantities. More elaborate formulae are postponed until Section 3 where the simulations are discussed. In general $\overline{T}_i$ and $\overline{T}_u$ are much smaller than T_L and T_t because of the incompatible speeds of CPU and peripheral devices. The ratio between C_{ch} and C_{CPU} is determined by the accounting policies and the characteristics of computer resource utilization. In a computation-bound system for example where most programs focus on CPU computations with few I/O accesses, C_{CPU} may be much larger than C_{ch}. The opposite may be true in an I/O-bound computer system.

3.1.2 Querying Cost for Multilist Organizations

In an inverted list organization, the lists corresponding to the keys in a query are accessed and manipulated so that one one must access only that set of data records which satisfies the query criterion exactly. Such a pre-search process is not possible for multilist file organizations, because the sets of record identifiers previously factored out as inverted lists are now distributed among the data records themselves (see Figure 2.8). The most commonly adopted search procedure for multilist organization is the following : For each product P_i ($= K_{i,1} \wedge K_{i,2} \wedge \cdots \wedge K_{i,n_i}$) in a given query Q, one selects from the n_i keys a key which appears in the least number of data records; every data record in the list for the selected key is then accessed. Following the adopted convention that $\ell_{i,1} \leq \ell_{i,2} \leq \cdots \leq \ell_{i,n_i}$ for each i, the selected key for P_i is always denoted as $K_{i,1}$. Since the query Q is a disjunction of the m products P_1, P_2, $\cdots$, and P_m, one has to access every data record in the lists corresponding to the m keys $K_{1,1}$, $K_{2,1}$, $\cdots$, and $K_{m,1}$. In summary, the above procedure can be rewritten as follows :

<u>Algorithm M</u> (Querying Procedure in Multilist Organization)

<u>Step 1</u> : Repeat step 2 for i = 1,2,...,m.

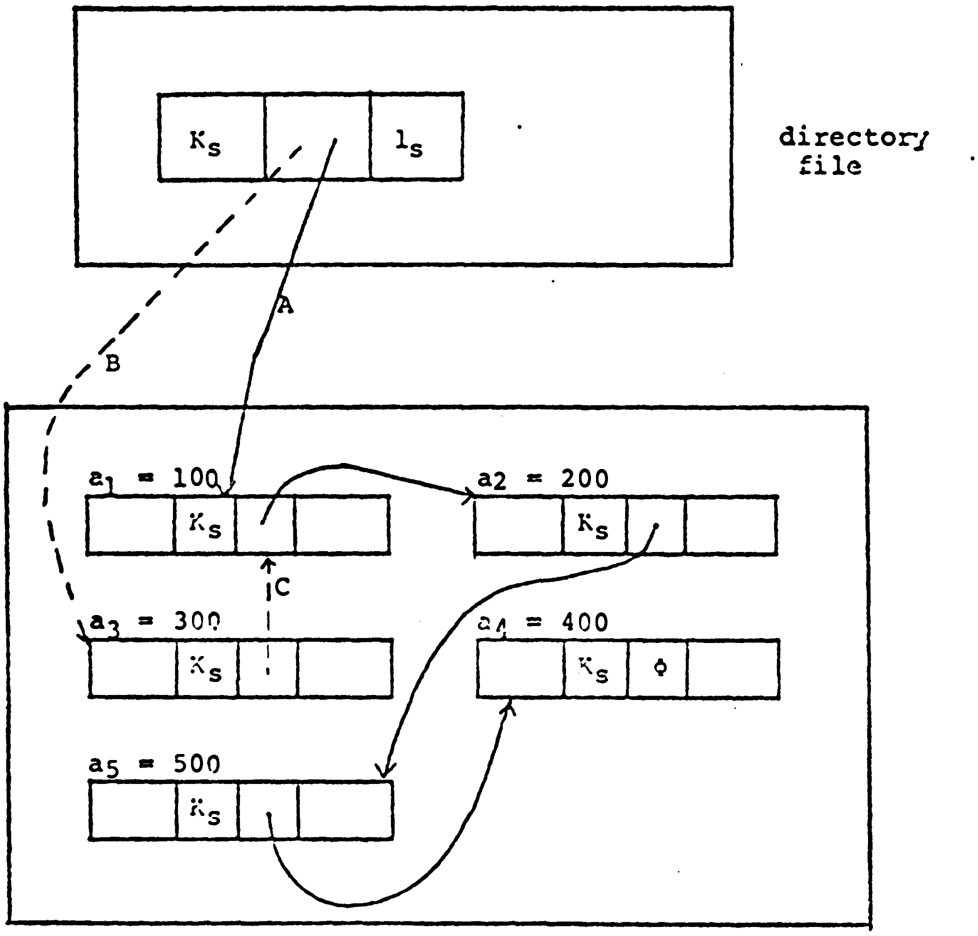
<u>Step 2</u> : Decode the n_i keys, $K_{i,1}$, $K_{i,2}$,...., and K_{i,n_i}. Select from these n_i keys a key which appears in the least number of data records (key $K_{i,1}$ by convention).

<u>Step 3</u> : Access every data record in the m lists corresponding to key $K_{1,1}$, $K_{2,1}$, ..., and $K_{m,1}$, and check each record against the query criterion.

Steps one and two simply decode all the keys in the query; the associated cost is $\sum_{i=1}^{m} n_i C_{decode}$. Step three is equivalent to accessing all the data records satisfying $K_1 \vee K_2 \vee \cdots \vee K_m$ (The second subscript of $K_{i,1}$ is dropped for simplicity). It should be noted that a record may appear in more than one of the multilists corresponding to the m keys, and one should avoid accessing such a record repeatedly. An efficient implementation of step three thus amounts to a solution of the following problem :

" Given m multilists corresponding to keys K_1, K_2, ..., K_m, how can one access each of the records in the m lists once and only once? "

A simple solution exists if the multilists are sorted in a certain order, say in increasing order of the record identifiers. Hsiao and Harary (1970) proposed an algorithm which would propagate simultaneously along the m multilists by accessing, at each step, the record with the smallest identifier value within the m lists. This clearly avoids duplicate accessing to the same record. Unfortunately, the file maintenance tasks are much harder to perform in sorted multilists than in unsorted multilists. Consider a simple addition of key K_s to a record identified by a_i in an unsorted multilist organization : it suffices to decode K_s in the directory, obtain the list head of the multilist chain for K_s, and then add record a_i as the new head of the multilist (Figure 3.1.a). If, however, the multilists are sorted, say by record identifier, then record a_i must be inserted at a specific point within the multilist chain so as to preserve the sorting order of chains. Such an insertion requires on average the traversal of half of the records in the multilist, and hence renders the update much



The a_i are record identifiers. After adding key K_s to record a_3, link A is replaced by links B and C and record a_3 becomes the new list head.

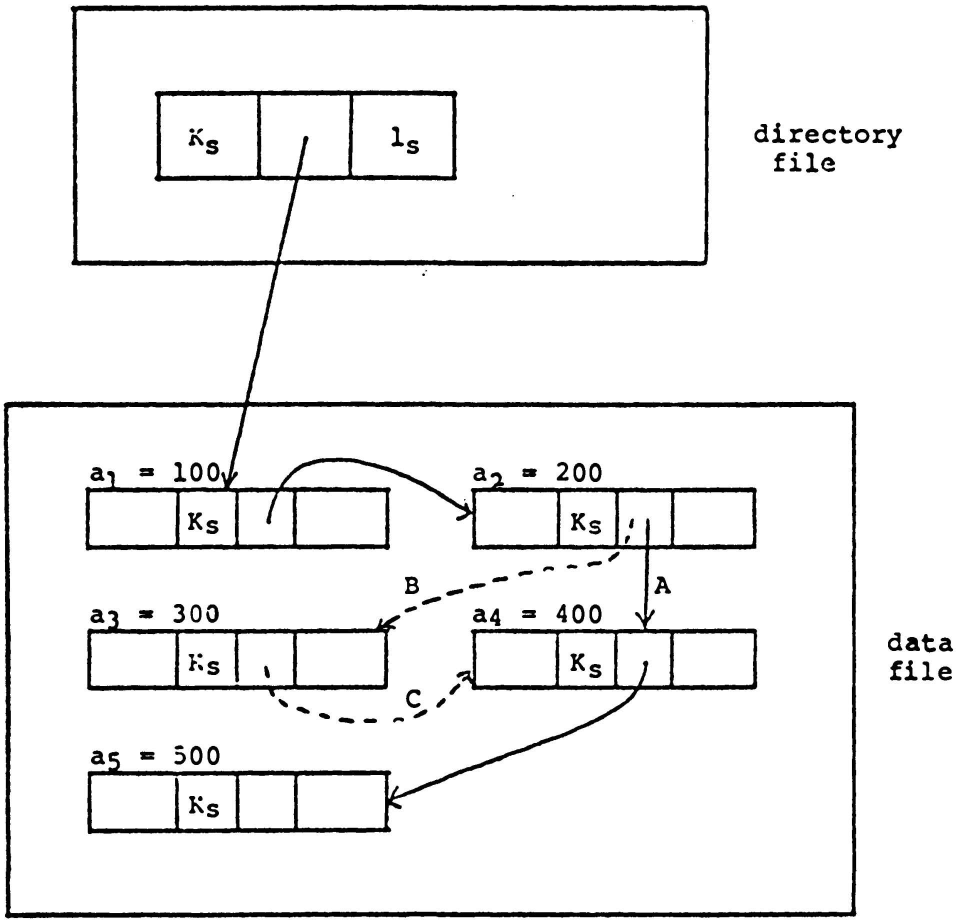
<u>Figure 3.1.a</u> Adding a key to a record when multilists are unsorted.

more expensive to perform (Figure 3.1.b). Accordingly, it is desirable to use a process which does not require sorted lists. The remaining part of this section designs a solution as such.

Consider the unsorted multilists corresponding to keys K_1, K_2, and K_3 in Figure 3.2. Record 30 contains all three keys and each of records 10 and 150 contains two of the three keys. If one accesses all the records in the K_1 list followed by all the records in K_2 and K_3 lists, record 30 will be retrieved three times and records 10 and 150 will each be retrieved twice. However, after a record identified as r is brought into main memory, one can easily check to see if this record also contains the rest of the two keys, K_1 and K_2. If record r indeed contains, say, key K_2, one can store an entry (2, r, s) in an internal table to indicate that record r in the K_2 list has already been examined and record s is the immediate successor of record r in the K_2 list. After all the records in the K_1 list are processed, one starts to retrieve the records in the K_2 list. Before actually issuing an I/O command to access any record, say identified by r, in the K_2 list, the internal table should be looked up for entry (2, r, s) for any s*. If such an entry is found in the table, record

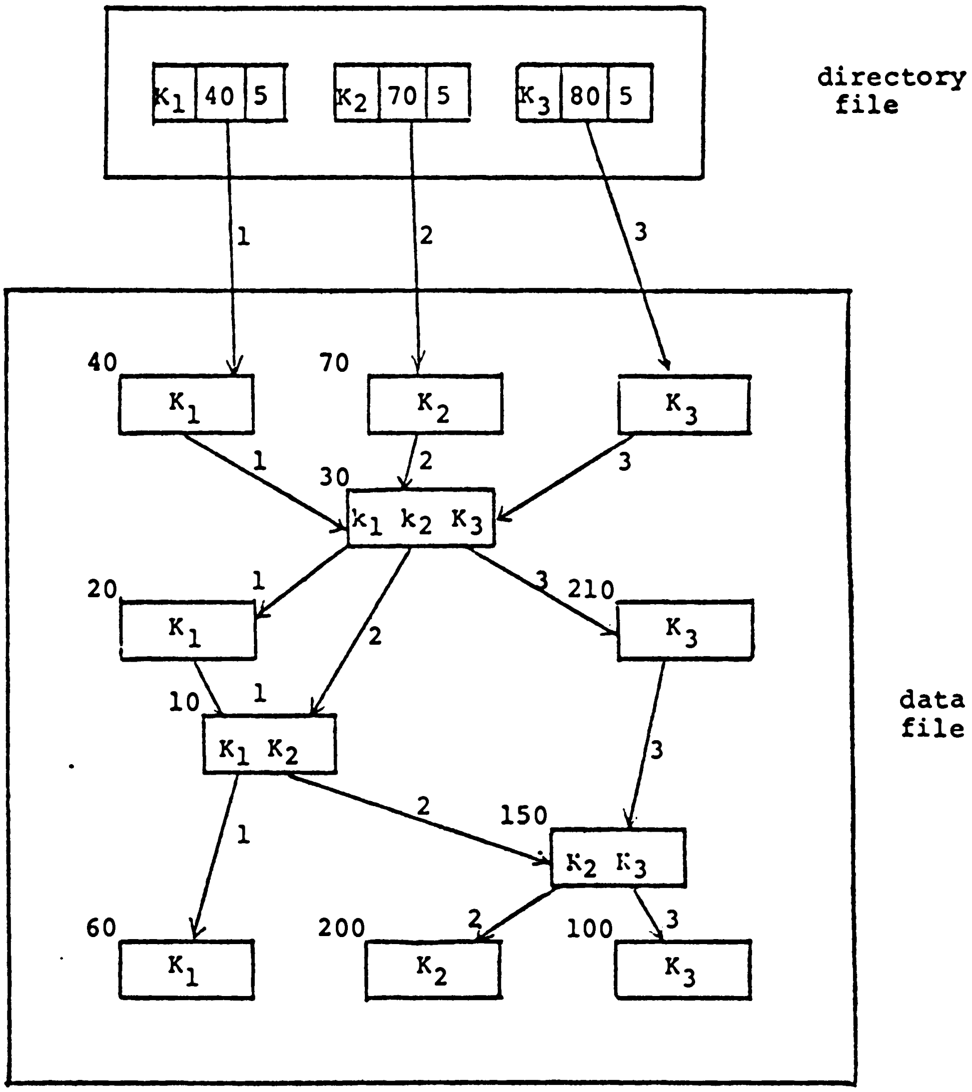
---

*Actually, there exists at most one such entry in the table since (2, r, s) indicates that record s is the immediate successor of record r in the K_2 list.



The a_i are record identifiers. After adding K_s to record a_3, link A is replaced by links B and C. Records a_1 and a_2 have to be accessed to update htese linkages.

<u>Figure 3.1.b</u> Adding a key to a record when multilists are sorted.



The number on the upper left hand corner of each record represents the record identifier. An edge labelled i indicates linkage in the K_i list.

Figure 3.2 An unsorted multilist organization for three keys.

r need not be accessed any more and record s becomes the next item to be considered for retrieval.

This process is illustrated in Figure 3.3 using the three multilists of Figure 3.2. After record 30 is accessed through the K_1 list, entries (2, 30, 10) and (3, 30, 210) are inserted into the table of Figure 3.3a. A third entry (2, 10, 150) is later inserted into the table when K_2 is found to exist in record 10.

After accessing records 40, 30, 20, 10, 60 through the K_1 list, the first record, 70, in the K_2 list is retrieved since no entry (2, 70, s) can be found in the table. The linkage information stored in record 70 should now indicate that its immediate K_2 succesor is record 30. Inasmuch as the entry (2, 30, 10) is found in the table, a new access to record 30 can be avoided and the next item to be considered for retrieval is record 10 as dictated by the entry (2, 30, 10). Record 10 also happens to have been retrieved already; the next record actually brought into main memory is thus record 150. After the retrieval of record 150, a new entry (3, 150, 100) is inserted into the table because this record also contains key K_3. Figure 3.3b shows the table content after all records in the K_1 and K_2 lists have been processed. At this point, the left part of the table in Figure 3.3b can actually be disgarded since the K_2 list has already been traversed completely. The remainder of the

(2, 30, 10)	(3, 30, 210)
(2, 10, 150)	

Figure 3.3.a Table content after records in the K_1 list of Figure 3.2 are accessed.

(2, 30, 10)	(3, 30, 210)
(2, 10, 150)	(3, 150, 100)

Figure 3.3.b Table content after records in the K_1 and K_2 lists of Figure 3.2 are accessed.

records that must actually be accessed are 80, 210, and 100.

Algorithm U below describes the above procedure formally. The internal table is implemented as a two dimensional array A where each element A(i,j) stores a two-tuple (r, s) instead of the 3-tuple (i, r, s) used in the previous illustration. Given m keys $K_1, K_2, \cdots, K_m$, all the records in the first list K_1 are accessed and an entry(r,s) is inserted into row A(i, $\cdot$) whenever record r is found to contain key K_i for $2 \leq i \leq m$. After all records in the K_1 list are accessed, the elements in A(i, $\cdot$) (i.e., A(i,1), A(i,2), ...) are sorted so that if A(i,j) = (r_j, s_j) and A(i, j+1) = (r_{j+1}, s_{j+1}) then $r_j < r_{j+1}$. This sorting facilitates the fast retrieval, from A(i, $\cdot$), of an element (r_x, s_x) whose first component r_x matches a given search argument r. After the sorting, the K_2 list is traversed and a record r is actually accessed only when an element of the like (r, s_x) can not be found in A(2, $\cdot$). In summary, the algorithm proceeds by traversing the K_i list followed by a sorting of array A(i+1, $\cdot$), for all i, until the last list corresponding to K_m is traversed.

<u>Definition</u> $S(r, K_i)$ = s if record s is the immediate successor of record r in the K_i list.

<u>Algorithm U</u> (Given m unordered multilists corresponding to keys $K_1, K_2, \cdots, K_m$, access each record which satisfies $K_1 \vee K_2 \vee \cdots \vee K_m$ once and only once)

<u>Step 0</u> : Initialize arrays $A(2,\cdot)$, $A(3,\cdot)$, $\cdots$, $A(m,\cdot)$ to empty.

<u>Step 1</u> : Repeat steps 2 through 8 for $i = 1, 2, \cdots, m$. (Steps 2 to 7 traverse the K_i list, and step 8 sorts array $A(i+1,\cdot)$)

<u>Step 2</u> : $r \leftarrow$ first record on K_i list.

<u>Step 3</u> : If $r = \phi$ then go to step 8(i.e. end of K_i list reached).

<u>Step 4</u> : If $i = 1$ then go to step 6 otherwise go to step 5.

<u>Step 5</u> : (Check if record r has been accessed before) If (r, s_x) is an element of $A(i,\cdot)$ for some s_x then $r \leftarrow s_x$ and go to step 3, otherwise to to step 6.

<u>Step 6</u> : (Access record r and insert array entries whenever necessary) Access record r. If record r contains K_n for $i+1 \leq n \leq m$ then insert entry $(r, S(r, K_n))$ to $A(n,\cdot)$.

<u>Step 7</u> : $r \leftarrow S(r, K_i)$, go to step 3.

<u>Step 8</u> : (Sort array $A(i+1,\cdot)$) If $i = m$ then STOP, otherwise sort $A(i+1,\cdot)$ such that if $A(i+1,j) = (r_j, s_j)$ and $A(i+1,j+1) = (r_{j+1}, s_{j+1})$ then $r_j < r_{j+1}$ for all j.

The proposed method avoids duplicate accesses to the same record at the expense of running Algorithm U in the central processing unit. Such a tradeoff between CPU processing and I/O accessing is faborable only if the running time of Algorithm U is low. The following paragraph shows that Algorithm U does have a small time bound in the worst case.

Running Time of Algorithm U

Let ℓ_i be the number of records containing key K_i. Since each element A(i,j) is defined to be (r_j, s_j) where record s_j is the immediate successor of record r_j in the K_i list, all element in array A(i,·) are distinct. Hence, the number of elements in A(i,·) is at most ℓ_i. Each execution of steps 0,1,2,3,4,6 and 7 takes constant time. One execution of step 5 takes at most $O(\log \ell_i)$operations if a binary search is performed on the sorted array A(i,·). In traversing the K_i list (i > 2), step 5 will be executed at most ℓ_i times, i.e. at most $O(\ell_i \log \ell_i)$ operations are performed. The sorting in step 8 is conducted once for each of the m-1 arrays, A(2,·), A(3,·), ···, and A(m,·). An $O(\ell_i \log \ell_i)$ sorting algorithm (say, heap sort or merge sort) can be adopted to implement step 8. Thus, the number of CPU operations is bounded by $O(\sum_{i=2}^{m} \ell_i \log \ell_i)$ for searching at step 5, and also $O(\sum_{i=2}^{m} \ell_i \log \ell_i)$ for sorting at step 8.

Hence, the worst case CPU time bound for Algorithm U is of order $O(\sum_{i=2}^{m} \ell_i \log \ell_i)$.

Since the CPU operations are much faster to perform then I/O operations, and Algorithm U has a low time bound in the worst case, it is reasonable to ignore the running time due to Algorithm U in computing the querying cost for multilist organization.

The cost calculation of Algorithm M for query processing in a multilist organizations can now be completed : Steps 1 and 2 of Algorithm M require $\sum_{i=1}^{m} n_i C_{decode}$ operations to decode all the keys in the query. Step 3 applies Algorithm U to control the access without duplication of w distinct records satisfying $K_1 \vee K_2 \vee \cdots \vee K_m$; the cost is $w (T_L + R T_t)$.

The total querying cost for processing

$$Q = \bigvee_{i=1}^{m} (K_{i,1} \wedge K_{i,2} \wedge \cdots \wedge K_{i,n_i})$$ in multilist

organizations is then :

$$QC_M = \sum_{i=1}^{m} n_i C_{decode} + w (T_L + R T_t) C_{ch}$$

3.2 Updating Costs for Inverted List and Multilist Organizations

The file maintenance operations account for another important part of the processing requirements in an information system. This section formulates cost equations for the following five typical on-line file maintenance operations in the environment of inverted list and multilist organizations :

1. add a key to a record,
2. delete a key from a record,
3. modify non-key data fields in a record,
4. add a new record to the file,
5. delete an old record from the file.

Data records in inverted list and multilist organizations are augmented by "deletion flags" (Figure 3.4) to facilitate file updating (Lefkovitz 1969). A one bit record deletion flag is attached to each record in both organizations. For multilist organizations, each key field in the data record is additionally associated with a one bit key deletion flag. If a record has a zero value in its record deletion flag, this means that the record is in the file; otherwise the record is considered to have been deleted from the file. The key deletion flags are set in a similar fashion.

The record deletion flags allow a logical deletion of records from a file. The physical removal of records is then postponed until file reorganization time. Key deletion flags are also used to erase keys logically. If a key together with the associated pointer is physically deleted from a record, the multilist chain of the corresponding key is broken. An alternative method is to connect the preceding record to the next one; this can not be efficiently implemented

record dele- tion flag	key field 1	key field 2	....	data field 1	data field 2	....

data record configuration in inverted list organizations

record dele- tion flag	key field 1	pointer 1	key dele- tion flag 1	key field 2	....	data feild 1	data filed 2	.....

data record configuration in multilist organizations

Figure 3.4 Data record configurations

unless two-way lists are used.

3.2.1 Updating Cost for Inverted List Organizations

Procedures and costs analyses for the five types of updata operations are described in this section. To avoid unnecessary tediousness, assume that each key appears in $\bar{I}$ data records on average, hence each inverted list is on the average $\bar{I}$ A bytes in length where A is the number of bytes needed to represent a record identifier. Also assume that each data record contains k keys on average.

1. Add key K to record r

   i. Decode key K from the directory; cost = C_{decode}.

   ii. Modify the directory entry of key K by incrementing the value of the length attribute by one, let this cost be defined to be C_K.

   iii. Access the inverted list corresponding to key k, insert record identifier r to the list and rewrite it to the list file;

   cost = $2\ (T_L + A\ \bar{I}\ T_t)\ C_{ch}$.

   iv. Access record r; insert K to r and rewrite it;

   cost $2\ (T_L + R\ T_t)\ C_{ch}$.

By i, ii, iii, and iv, the total cost to add a key to a record is therefore :

$$UC_{I1} = C_{decode} + C_K + [4\ T_L + 2\ (A\ \bar{I} + R)\ T_t]\ C_{ch}$$

2. <u>Delete key K from record r</u>

This procedure is exactly paralled to that in 1. The cost is also the same as UC_{I1}, i.e., $UC_{I2} = UC_{I1}$.

3. <u>Modify non-key data fields in record r</u>

Access record r; modify the data fields and rewrite it to data file.

$UC_{I3} = 2\ (T_L + R\ T_t)\ C_{ch}$

4. <u>Add a new record of k keys to the file</u>

Adding a new record of k keys is conceptually equivalent to adding k keys to an existing record; the procedure for type 4 updates is then similar to that for type 1 updatas. The cost UC_{I4} can thus be easily assessed by referring to the way in which UC_{I1} was formulated earlier.

$$UC_{I4} = k\ (C_{decode} + C_K) + 2\ (k\ T_L + k\ \overline{\Sigma}\ A\ T_t)\ C_{ch} + (T_L + R\ T_t)\ C_{ch}.$$

5. <u>Delete an old record from the file</u>

Access the record, set record deletion flag to 1 and rewrite the record into the data file.

$UC_{I5} = 2\ (T_L + R\ T_t)\ C_{ch}$.

Let P_i be the probability that an update operation is of type i, then the total average cost of an update operation can be written as :

$$UC_I = \sum_{i=1}^{5} P_i\ UC_{Ii},$$

3.2.2 Updating Cost for Multilist Organizations

1. Add key K to record r

   As described in Figure 2.8, the directory entry for key K contains three components (K, P_K, ℓ_K) where P_K points to the first record in the list and ℓ_K refers to the number of records in the list. The procedure for adding key K to record r is to decode K first, and then to replace entry (K, P_K, ℓ_K) by $(K, r, \ell_K + 1)$. Finally, insert P_K into the pointer subfield associated with key K in record r, and rewrite r into the data file. This effectively makes record r the new head of list K. A multilist is shown in Figure 3.5 before and after a type one update. The type 1 update cost is then :

   $UC_{M1} = C_{decode} + C_K + 2\ (T_L + R\ T_t)\ C_{ch}$

2. Delete key K from record r

   Retrieve record r, set the key deletion flag associated with key K to 1, then rewrite record r. Cost UC_{I2} is equal to $2\ (T_L + R\ T_t)\ C_{ch}$.

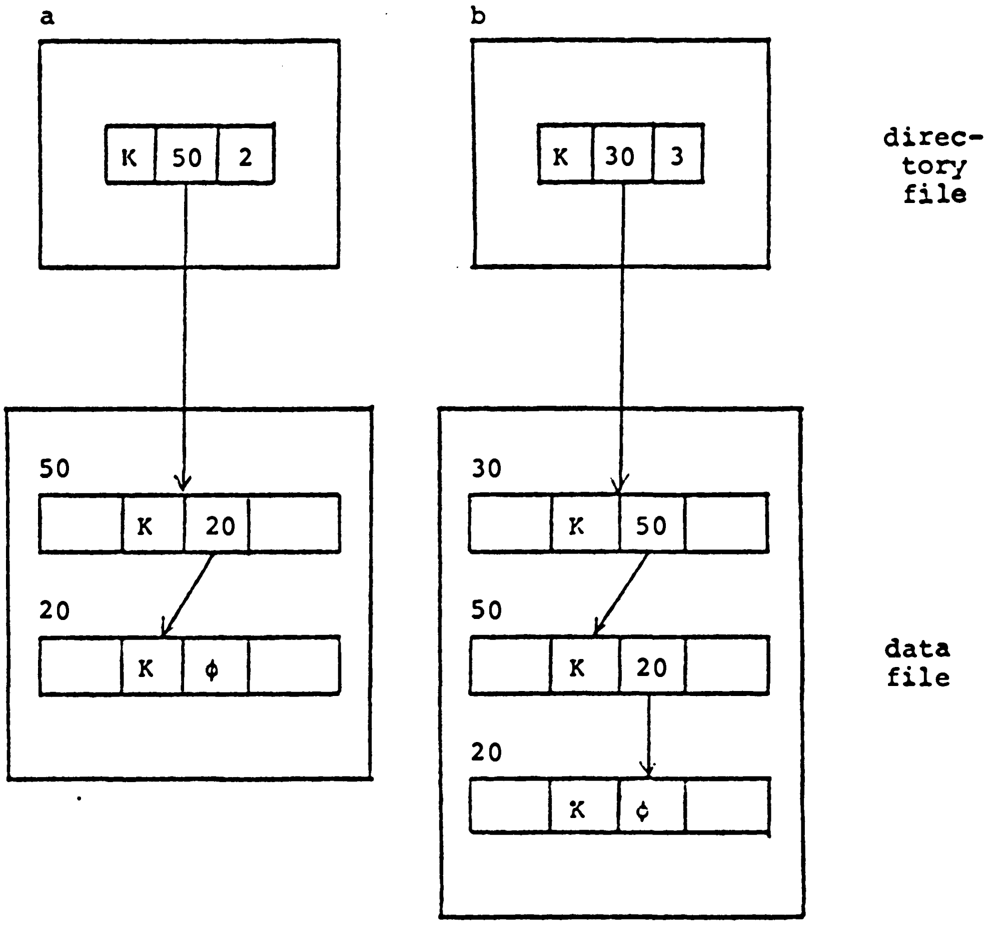
3. Modify non-key data fields in record r

   This again involves only retrieving, modifying, and rewriting record r. So $UC_{M3} = 2\ (T_L + R\ T_t)\ C_{ch}$.

4. Add a new record of k keys to the file

   By an argument similar to that for type 1 updates, UC_{M4} can be written as

   $UC_{M4} = k\ (C_{decode} + C_K) + (T_L + R\ T_t)\ C_{ch}$



Diagrams a and b represent the multilist organization corresponding to key K before and after key K is added to record 30.

Figure 3.5 Type 1 update in a multilist organization

5. Delete an old record from the file

Access the record, set the record deletion flag to 1 and rewrite this record.

$UC_{M5} = 2\ (T_L + R\ T_t)\ C_{ch}$.

The total average cost of an update operation in a multilist is $UC_M = \sum_{i=1}^{5} P_i\ UC_{Mi}$.

The update costs for inverted list and multilist organization are summarized in Figure 3.6. For type 3 and 5 updates, both organizations have the same cost. For each of the remaining three update types, the multilist organization is less expensive to update than the inverted list organization. This is because the inverted lists themselves must also be modfied in the inverted list organization in addition to the data files.

3.3 Structure and Processing Algorithm for a Class of Hybrid List Organizations

Unsorted multilist organizations were shown to be more efficient than inverted list organizations for file maintenance. It is interesting to compare the query processing efficiency for these two organizations. The cost equations for processing a query of the form $\bigvee_{i=1}^{m} (K_{i,1} \wedge K_{i,2} \wedge \cdots \wedge K_{i,n_i})$ are reporduced as follows :

	Organization	
Types of update operations	Inverted list	Multilist
1	$UC_{I1} = C_{decode} + C_k$ $+[4\ T_L + 2(\Lambda\ \bar{I} + R)\ T_t]\ C_{ch}$	$UC_{M1} = C_{decode} + C_k$ $+2\ (T_L + R\ T_t)\ C_{ch}$
2	$UC_{I2} = C_{decode} + C_k$ $+[4\ T_L + 2(\Lambda\ \bar{I} + R)\ T_t]\ C_{ch}$	$UC_{M2} = 2(T_L + R\ T_t)\ C_{ch}$
3	$UC_{I3} = 2(T_L + R\ T_t)\ C_{ch}$	$UC_{M3} = 2(T_L + R\ T_t)\ C_{ch}$
4	$UC_{I4} = k(C_{decode} + C_k)$ $+2(k\ T_L + k\ \bar{I}\ A\ T_t)\ C_{ch}$ $+(T_L + R\ T_t)\ C_{ch}$	$UC_{M4} = k(C_{decode} + C_k)$ $+(T_L + R\ T_t)\ C_{ch}$
5	$UC_{I5} = 2(T_L + R\ T_t)\ C_{ch}$	$UC_{M5} = 2(T_L + R\ T_t)\ C_{ch}$

<u>Figure 3.6</u> Cost comparisons for the five types of update operations in inverted list and multilist organizations.

$$QC_I = \left(\sum_{i=1}^{m} n_i\right) C_{decode} + \sum_{i=1}^{m} \left(n_i T_L + \sum_{j=1} \ell_{i,j} A T_t\right) C_{ch}$$

$$+ \left(\sum_{i=1}^{m} (n_i - 1) \overline{T}_i + (m - 1) \overline{T}_u\right) C_{CPU}$$

$$+ \ell (T_L + R T_t) C_{ch}$$

$$QC_M = \left(\sum_{i=1}^{m} n_i\right) C_{decode} + w (T_L + R T_t) C_{ch}.$$

These equations are too complicated to permit meaningful analysis. One can gain more insight by starting with cost equations for simpler queries. Consider a query consisting of only one product of keys, $Q = K_{i,1} \wedge K_{i,2} \wedge \cdots \wedge K_{i,n_i}$. The cost equations for such a query are denoted as QC_{IS} and QC_{MS}* and can be rewritten as

$$QC_{IS} = n_i C_{decode} + \left(n_i T_L + \sum_{j=1}^{n_i} \ell_{i,j} A T_t\right) C_{ch}$$

$$+ (n_i - 1) \overline{T}_i C_{CPU} + \ell_i (T_L + R T_t) C_{ch}$$

$$QC_{MS} = n_i C_{decode} + \ell_{i,1} (T_L + R T_t) C_{ch}$$

To simplify the situation even further, one can ignore for the time being the $(n_i - 1) \overline{T}_i C_{CPU}$ term in QC_{IS} since $\overline{T}_i$, the average amount of time to perform one set intersec-

---

* The subscript "S" stands for "simplified".

tion operation in CPU, is much smaller than T_L and T_t, the device latency and transmission times. Subtracting QC_{IS} from QC_{MS} one obtains

$$QC_{MS} - QC_{IS} = (\ell_{i,1} - \ell_i - n_i)\ T_L\ C_{ch}$$

$$+ (\ell_{i,1}\ R - \ell_i\ R - \sum_{j=1}^{n_i} \ell_{i,j}A)\ T_t\ C_{ch}$$

Note that $\ell_{i,1}$ is the number of data records in which key $K_{i,1}$ appears, and ℓ_i is the number of data records which satisfy $K_{i,1} \wedge K_{i,2} \wedge \cdots \wedge K_{i,n_i}$; hence $\ell_{i,1}$ is larger than or equal to ℓ_i. Equation $QC_{MS} - QC_{IS}$ does not indicate either that QC_{MS} is always smaller than QC_{IS} or the opposite. Consider the situation where $\ell_{i,1}$ is very large. The difference between $\ell_{i,1}$ and ℓ_i may then be large or small depending on how close ℓ_i is to $\ell_{i,1}$. In practice, when $\ell_{i,1}$ is large, $\ell_{i,1} - \ell_i$ tends to be large enough so that both $\ell_{i,1} - \ell_i - n_i$ and $\ell_{i,1}\ R - \ell_i\ R - \sum_{j=1}^{n_i} \ell_{i,j}\ A$ are both likely to be positive. This implies that QC_{MS} is frequently larger than QC_{IS}, in which case the multilist organization is more costly.

On the other hand, if $\ell_{i,1}$ is small, the difference between $\ell_{i,1} - \ell_i$ is even smaller. As a result of this, $\ell_{i,1} - \ell_i - n_i$ and $\ell_{i,1}\ R - \ell_i\ R - \sum_{j=1}^{n_i} \ell_{i,j}\ A$ are likely to assume negative values and the inverted list organization becomes more expensive for query processing.

This analysis immediately suggests that a file structure which would process a query according to the procedure used in inverted list organization when $\ell_{i,1}$ is large, while using an organization similar to a multilist structure when $\ell_{i,1}$ is small, would outperform both pure inverted list organization and pure multilist organizations. To achieve this purpose, a hybrid list organization is defined as follows :

<u>Definition</u> A <u>hybrid list organization of parameter</u> ℓ_{th} is a list structure which stores the set of identifiers for the $\ell_{i,j}$ records containing key $K_{i,j}$ as a separate inverted list if $\ell_{i,j}$ is larger than ℓ_{th}; otherwise the set of identifiers is embedded in the data records as a multilist organization.

The hybrid list organization actually incorporates a whole class of list file structures. Each element in the class is characterized by a specific value of parameter ℓ_{th}. When ℓ_{th} is larger than zero and less than infinity, the corresponding hybrid organization typically contains a non-zero number of multilists and a non-zero number of inverted lists and is called a <u>non-degenerate</u> hybrid list organization. In the special case where ℓ_{th} is equal to zero, all lists are implemented as inverted lists and the hybrid organization degenerates into a pure inverted list organization. At the other extreme, when ℓ_{th} is equal to infinity, a pure multilist organization is produced. As ℓ_{th} increases from zero to infinity, more and more lists are stored as multilists.

Querying Procedure for Hybrid List Organizations

The querying procedure for a non-degenerate hybrid list organization is a combination of the algorithm used for the two pure list structures. For each product $P_i = K_{i,1} \wedge K_{i,2} \wedge \cdots \wedge K_{i,n_i}$ in query Q ($= \bigvee_{i=1}^{m} P_i$), the product P_i is tackled by using the algorithm used for inverted lists if $\ell_{i,1}$ (note the convention that $\ell_{i,1} \leq \ell_{i,2} \leq \cdots \leq \ell_{i,n_i}$) is larger than ℓ_{th}. If instead $\ell_{i,1}$ is smaller than ℓ_{th}, product P_i will be processed through the multilist procedure, i.e. by accessing every record in the $K_{i,1}$ list. To answer the whole query (which is a sum of the individual products) one simply uses Algorithm M of section 3.1.2 to traverse the multilists selected for some of the products, followed by an access to the records presented by the inverted lists corresponding to the rest of the products, assuming that these records have not been retrieved through the multilists.

The proposal for the class of hybrid list organization has been based on observations relating to the nature of pure inverted list and multilist organizations. No evidence has been given as yet to assure that the non-degenerate hybrid structures are worthwhile. Neither does one know how the performance of these organizations is affected by the parameter ℓ_{th}. Questions of this sort are investigated in the next two sections.

File Maintenance Procedure for Hybrid List Organizations

File maintenance in a hybrid list organization also alternates between the procedures for the two pure organizations. If a key is involved in a modification and its associated key list is stored as a multilist, then the multilist maintenance procedure is adopted; otherwise the key list is modified as an inverted list.

Since it is easier to modify a multilist than an inverted list as shown in section 3.2.2, the cost of file maintenance for a non-degenerate hybrid structure lies somewhere between the costs for the two degenerate cases. Of course, the more lists are stored as multilists, the easier it is to update the file. Simulation results are presented in section 3.5 to show that non-degenerate hybrid list organizations may actually outperform both pure structures when both querying and updating costs are counted.

3.4 A Simplified Analysis of Querying Cost for Hybrid List Organizations

The effectiveness of any file organization is highly dependent on the type of system environment in which it is used. An information system is usually characterized by a set of parameters which describe the properties of queries, update operations, file elements, and storage devices, etc.. In this section, a set of simplified assumptions is adopted to describe the queries and files in the system. Using such a simple model, analytic results are presented to show that, by a proper choice of the ℓ_{th} values, the non-degenerate hybrid list organization may indeed be superior to both inverted list and multilist organizations.

Simplified System Model

The simplified system model is based on the following assumptions :

Assumption 1 Every query consists of only one product of keys, that is, $Q = K_{i,1} \wedge K_{i,2} \wedge \cdots \wedge K_{i,n_i}$*. Assume that key $K_{i,j}$ appears in $\ell_{i,j}$ data records and that $\ell_{i,1} \leq \ell_{i,2} \leq \cdots \leq \ell_{i,n_i}$. Let $f_K(n_i)$ be the probability density function of n_i for $n_\ell \leq n_i \leq n_u$. The average value of n_i is denoted as $\bar{n} = E(n_i) = \sum_{n_i = n_\ell}^{n_u} n_i \, f_K(n_i)$.

---
* The subscript i is actually unnecessary for a query of one product only, but it is retained here to avoid confusion with the symbols used in the other sections of this chapter.

<u>Assumption 2</u> Let $\ell_{i,1}$ be distributed uniformly in the range of $[L_\ell, L_u]$, that is, the probability density function of $\ell_{i,1}$ is defined as

$f_\ell(\ell_{i,1}) = 1 / (L_u - L_\ell + 1)$ for $L_\ell \leq \ell_{i,1} \leq L_u$.

<u>Assumption 3</u> Let ℓ_i be the number of records containing keys $K_{i,1}$, $K_{i,2}$, ..., and K_{i,n_i} simultaneously. Let $\ell_i = \rho\ \ell_{i,1}$ for $0 \leq \rho \leq 1$. Note that $\ell_{i,1}$ is the number of records in which key $K_{i,1}$ appears. $\ell_i = \rho \cdot \ell_{i,1}$ implies that a fraction of the records containing $K_{i,1}$ also contains $K_{i,2}$, $K_{i,3}$, ..., K_{i,n_i}.

The costs to process a single query of the form $K_{i,1} \wedge K_{i,2} \wedge \cdots \wedge K_{i,n_i}$ for inverted list and multilist organizations respectively are

$$QC_{IS} = n_i\ C_{decode} + (n_i\ T_L + \sum_{j=1}^{n_i} \ell_{i,j}\ A\ T_t)\ C_{ch}$$

$$+ (n_i - 1)\ \overline{T}_i\ C_{CPU} + \ell_i\ (T_L + R\ T_t)\ C_{ch}$$

and

$$QC_{MS} = n_i\ C_{decode} + \ell_{i,1}\ (T_L + R\ T_t)\ C_{ch}.$$

Given a set of queries which are characterized by the three assumptions, the average costs to process such queries in the two organizations are then

$$E(QC_{IS}) = \sum_{n_i=n_\ell}^{n_u} f_K(n_i)\ (n_i\ C_{decode} + (n_i - 1)\ \overline{T}_i\ C_{CPU})$$

$$+ \sum_{n_i=n_\ell}^{n_u} \sum_{\ell_{i,1}=L_\ell}^{L_u} f_K(n_i)\, f_\ell(\ell_{i,1}) \cdot$$

$$[(n_i + \rho\, \ell_{i,1})\, T_L + (A\, n_i\, \bar{\ell} + \rho\, \ell_{i,1}\, R)\, T_t]\, C_{ch}$$

and

$$E(QC_{MS}) = \sum_{n_i=n_\ell}^{n_u} f_K(n_i)\, n_i\, C_{decode}$$

$$+ \sum_{n_i=n_\ell}^{n_u} \sum_{\ell_{i,1}=L_\ell}^{L_u} f_K(n_i)\, f_\ell(\ell_{i,1}) \cdot$$

$$(\ell_{i,1}\, T_L + \ell_{i,1}\, R\, T_t)\, C_{ch}$$

Note that $\bar{\ell}$, the average number of data records in which a key appears, is used in the $E(QC_{IS})$ equation.

If a hybrid list organization with parameter ℓ_{th} is adopted, queries with $\ell_{i,1}$ larger than ℓ_{th} are processed as an inverted list organization, and queries with $\ell_{i,1} \leq \ell_{th}$ are processed by the multilist procedure. The average cost for hybrid organization is then

$$E(QC_{HS}) = \sum_{n_i=n_\ell}^{n_u} f_K(n_i)\, n_i\, C_{decode}$$

$$+ \sum_{n_i=n_\ell}^{n_u} \sum_{\ell_{i,1}=L_\ell}^{\ell_{th}} f_K(n_i)\, f_\ell(\ell_{i,1})\, (\ell_{i,1}\, T_L +$$

$$\ell_{i,1}\, R\, T_t)\, C_{ch}$$

$$+ \sum_{n_i=n_\ell}^{n_u} \sum_{\ell_{i,1}=\ell_{th}+1}^{L_u} f_K(n_i)\, f_\ell(\ell_{i,1}) \cdot$$

$$\{(n_i - 1)\ \overline{T}_i\ C_{CPU} + [(n_i + \rho\ \ell_{i,1})\ T_L + (A\ n_i\ \overline{\ell} + \rho\ \ell_{i,1}\ R)\ T_t]\ C_{ch}\}$$

Using these equations, lemma one derives a sufficient condition under which the hybrid organization is better than the inverted list organization. Lemma 2 shows similar results for a multilist organization.

<u>Lemma 1</u> $E(QC_{HS}) < E(QC_{IS})$ if $L_\ell - 1 < \ell_{th} < L_0 - L_\ell$

$$\text{where } L_0 = \frac{2\ [\overline{n}\ (T_L + A\ \overline{\ell}\ T_t) + (\overline{n} - 1)\ \overline{T}_i\ C_{CPU} / C_{ch}]}{(1 - \rho)\ (T_L + R\ T_t)}$$

(Proof)

$E(QC_{HS}) - E(QC_{IS})$

$$= \sum_{n_i=n_\ell}^{n_u} \sum_{\ell_{i,1}=L_\ell}^{\ell_{th}} f_K(n_i)\ f_\ell(\ell_{i,1})\ (\ell_{i,1}\ T_L + R\ \ell_{i,1}\ T_t) C_{ch}$$

$$- \sum_{n_i=n_\ell}^{n_u} \sum_{\ell_{i,1}=L_\ell}^{\ell_{th}} f_K(n_i)\ f_\ell(\ell_{i,1})\ \{(n_i - 1)\ \overline{T}_i\ C_{CPU}$$

$$+ [(n_i + \rho\ \ell_{i,1})\ T_L + (A\ n_i\ \overline{\ell} + \rho\ \ell_{i,1}\ R)] T_t\ C_{ch}\}$$

$$= \sum_{n_i=n_\ell}^{n_u} \sum_{\ell_{i,1}=L_\ell}^{\ell_{th}} f_K(n_i)\ f_\ell(\ell_{i,1})\ \{[\ell_{i,1}(1 - \rho) - n_i] \cdot$$

$$T_L + [R\ \ell_{i,1}\ (1 - \rho) - A\ n_i\ \overline{\ell}]\ T_t\}\ C_{ch}$$

$$- \sum_{n_i=n_\ell}^{n_u} \sum_{\ell_{i,1}=L_\ell}^{\ell_{th}} f_K(n_i)\ f_\ell(\ell_{i,1})\ (n_i - 1)\ \overline{T}_i\ C_{CPU}$$

$$= \sum_{n_i=n_\ell}^{n_u} f_K(n_i)\ \frac{1}{L_u - L_\ell + 1}\ \{[\frac{(L_\ell + \ell_{th})(\ell_{th} - L_\ell + 1)}{2} \cdot$$

$$(1 - \rho) - n_i\ (\ell_{th} - L_\ell + 1)]\ T_L$$

$$+[\frac{(L_\ell + \ell_{th})(\ell_{th} - L_\ell + 1)}{2}\ R\ (1 - \rho)$$

$$- A\ n_i\ \bar{\ell}\ (\ell_{th} - L_\ell + 1)]\ T_t\}\ C_{ch}$$

$$- \frac{\ell_{th} - L_\ell + 1}{L_u - L_\ell + 1}\ (\bar{n} - 1)\ \bar{T}_i\ C_{core}$$

$$= \frac{\ell_{th} - L_\ell + 1}{L_u - L_\ell + 1}\ \{[(L_\ell + \ell_{th})(1 - \rho)(T_L + R\ T_t) / 2$$

$$- \bar{n}\ (T_L + A\ \bar{\ell}\ T_t)]\ C_{ch} - (\bar{n} - 1)\ \bar{T}_i\ C_{CPU}\}$$

Hence a sufficient condition for $E(QC_{HS}) < E(QC_{IS})$ is that both $\ell_{th} - L_\ell + 1 > 0$ and $(L_\ell + \ell_{th})(1 - \rho)(T_L + R\ T_t) / 2 - \bar{n}\ (T_L + A\ \bar{\ell}\ T_t)\ C_{ch} - (\bar{n} - 1)\ \bar{T}_i\ C_{CPU} < 0$ are true. Combining both inequalities, one obtains the following condition

$$L_\ell - 1 < \ell_{th} < L_0 - L_\ell$$

Q. E. D.

Lemma 2 $E(QC_{HS})$ $E(QC_{MS})$ if $L_0 - L_u - 1 < \ell_{th} < L_u$ where L_0 assumes the same form as in lemma one.

The proof of lemma 2 is similar to that of lemma 1.

Lemma one implies that, in the simplified system model a hybrid list organization can outperform a pure inverted list organization for processing queries if the ℓ_{th} value characterizing the hybrid organization is in the range of

$[L_\ell - 1, L_0 - L_\ell]$. Lemma two specifies another ℓ_{th} range $[L_0 - L_u - 1, L_u]$ for which the corresponding hybrid list organization is more efficient than a pure multilist organization for query processing. A more interesting result than the previous two would be to find ℓ_{th} values which produce hybrid structures that are simultaneously better than pure inverted list as well as multilist organizations. Such ℓ_{th} values exist if the two intervals in lemma one and lemma two overlap.

It is a well known fact that two intervals, (a_1, b_1) and (a_2, b_2), overlap if and only if $b_1 > a_2$ and $b_2 > a_1$. The overlapping part is (c_1, c_2) where $c_1 = \max\{a_1, a_2\}$ and $c_2 = \min\{b_1, b_2\}$. Let $(L_\ell - 1, L_0 - L_\ell) = (a_1, b_1)$ and $(L_0 - L_u - 1, L_u) = (a_2, b_2)$. Clearly two intervals in lemma 1 and lemma 2 overlap since $b_2 - a_1 = L_u - L_\ell + 1 > 0$ and $b_1 - a_2 = L_u - L_\ell + 1 > 0$. The above argument establishes the following result :

<u>Lemma 3</u> (i) (c_1, c_2) is an non-empty interval, where $c_1 = \max\{L_\ell - 1, L_0 - L_u - 1\}$ and $c_2 = \min\{L_0 - L_\ell, L_u\}$.

(ii) $E(QC_{HS}) < E(QC_{IS})$ and $E(QC_{HS}) < E(QC_{MS})$ are both true if $c_1 < \ell_{th} < c_2$.

3.5 Simulation Studies for the Class of Hybrid List Organization

The results obtained in the previous section indicate that a non-degenerate hybrid structure may indeed be valuable for processing queries in a simplified model. It is the purpose of this section to extend these results to more realistic environments through simulation techniques. Section 3.5.1 describes a set of parameters which charaterize the system model. Simulation results on query processing are presented in section 3.5.2. Section 3.5.3 incorporates file maintenance operations into the model.

3.5.1 Parametric Description of the Simulation Models

The parpmeters which are related to file processing in an information system can be classified into three categoreis. The first category includes variables which sum up the specifications of the computing devices used in the system. The parameters in the second category are concerned with certain statistics of the records in the data base. The third category contains parameters that jointly characterize the prototypes of queries and updates submitted to the system. Most of these parameters have been included in Table 3.1 and they are edited in Table 3.2 according to the categories to which they belong.

The queries used in the simulation are in the their most general form, $\bigvee_{i=1}^{m} (K_{i,1} \wedge K_{i,2} \wedge \cdots \wedge K_{i,n_i})$. The cost equations for the inverted list and multilist organizations, Device related parameters : T_L, T_t, A, C_{decode}, C_K, C_{ch}, C_{CPU}

Data base related parameters : X, k, R

Querying and updating related parameters : m, n_i, $\ell_{i,j}$, ℓ_i, P_1, P_2, P_3, P_4, P_5, P_I, P_M, P_Q, P_V

Explanation of these symbols can be found in Table 3.1.

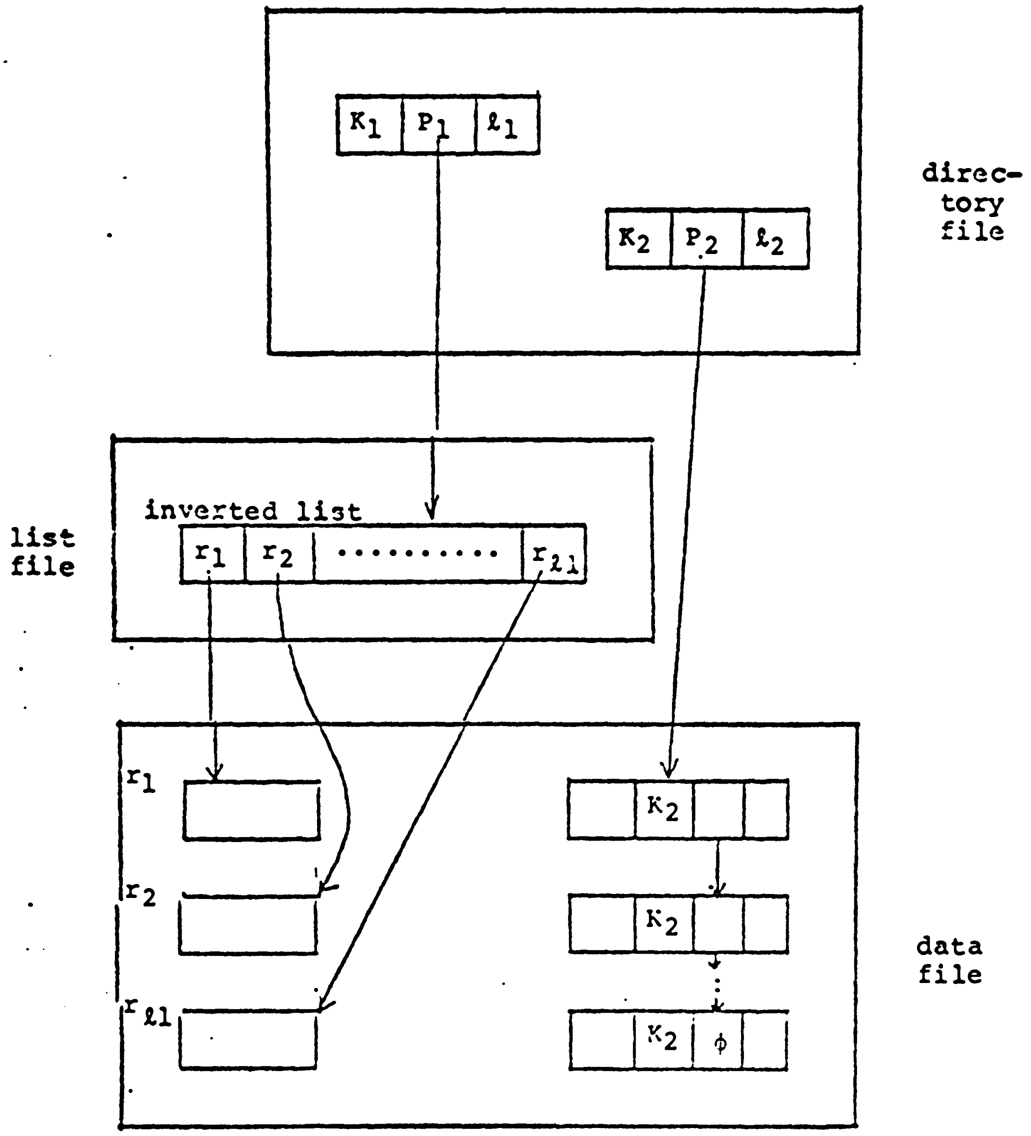
Table 3.2 Parameters used in the simulation model.

QC_I and QC_M, were formulated in sections 3.1.1 and 3.1.2 respectively. Figure 3.6 tabulates cost expressions for various file maintenance operations for the above two organizations. The querying and file maintenance procedures in section 3.3 indicate that the corresponding equations for the non-degenerate hybrid organizations can be expressed as a combination of the equations for the degenerate cases. The rest of this section determines the values of the variables included in these equations so that costs reflecting realistic system environments can be obtained for the whole class of hybrid list organizations. From now on, the phrase "hybrid list organizations" refers to both degenerate and non-degenerate hybrid list organizations. The structure of a hybrid organization is shown in Figure 3.7. Throughout the simulation process, it is assumed that an IBM 360/65 is the underlying computer system.

Device related parameters

1. T_L, T_t and A

Assume that the list and data files of a hybrid organization are stored on IBM 2314 disk packs. For such disk devices, T_L (average latency time) is equal to 12.5 msec, T_t (time needed to transfer one byte of data to main memory) is 0.00322 msec, and A (number of bytes needed to rerepresent an address on the device) is 8 bytes.



Note that $\ell_1 > \ell_{th}$ and $\ell_2 \leq \ell_{th}$

Figure 3.7 Structure of a hybrid list organization with parameter ℓ_{th}.

2. C_{decode} and C_K

The directory file is usually smaller and also more frequently accessed than the list and data files; hence it is reasonable to save the directory on a faster device, say, on an IBM 2305 drum which has an average latency time of 2.5 msec. C_K, the time needed to to restore a modified key entry to the directory file, is then equal to the 2.5 msec latency plus the time interval in which the key entry is actually being written. The latter portion is very small and can be ignored.

Cdecode, the average cost of decoding one key from the directory, depends on the type of organization adopted for the directory. An excellent organization which accomplishes fast retrieval and update operations in a direcotry is the B-tree structure proposed by Bayer and McCreight. Their experimental results show that a proper implementation of the B-tree would require only 0.83 accesses on average to the peripheral devices when several buffers are allocated in the main memory (Bayer and McCreight 1972). For the simulation purpose in this section, assume that B-tree is used for the directory and Cdecode is equal to the cost of one access to the drum.

3. C_{ch} and C_{CPU}

For the purpose of comparing the costs of the elements in the class of hybrid list organization, it is only necessary to determine the ratio of the two parameters, C_{ch} and C_{CPU}, or the costs of one unit of channel and CPU time. As pointed out at the end of section 3.1.1, the ratio between C_{ch} and C_{CPU} is affected by accounting policies and resource utilization characteristics of the computer system. A typical value of 10 is used for C_{CPU}/C_{ch} in this simulation.

Data base related parameters

1. X (list length, or the number of data records in which a key appears)

The distribution of list lengths X in a file may have critical effects on the performance of a specific file organization. The Beta and exponential distributions are the two probability density functions adopted to simulate the list lengths X. Both functions are reasonable descriptions of the list length distributions in some typical data bases. The curves for both functions can be tuned by some parameters to emphasize or deemphasize the probability densities at various X values. Such tunings can imitate different list length characteristics in different data bases. Peculiarities of beta and exponential distributions are summarized in the following two paragraphs.

Beta distribution

A beta distribution with parameters a and b is denoted as

$\beta(a,b)$. The probability density function of $\beta(a,b)$ is

$$f(X) = \frac{\Gamma(a+b)}{\Gamma(a)\,\Gamma(b)} X^{a-1} (1-X)^{b-1} \quad \text{for } 0 \leq X \leq 1$$

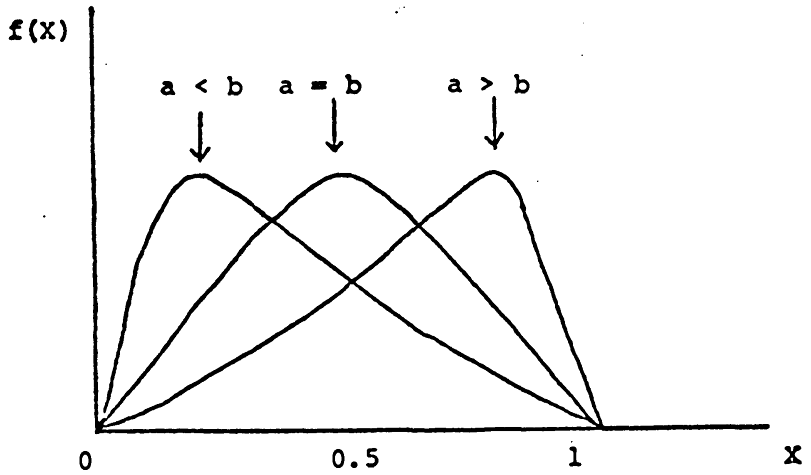
where $\Gamma(n) = (n-1)\Gamma(n-1)$ $(n \geq 1)$ and $\Gamma(0) = 1$.
The curve of $f(X)$ is a symmetric bell shape when $a = b$ and left or right biased if a is larger than or smaller than b (Figure 3.8). Although $f(X)$ is defined above for X between zero and one, the beta distribution can actually be defined on any interval by a linear transformation of the random variable X.

Five instances of the beta distribution for various parameter values and variable ranges are used to express the list length statistics in the data base; namely

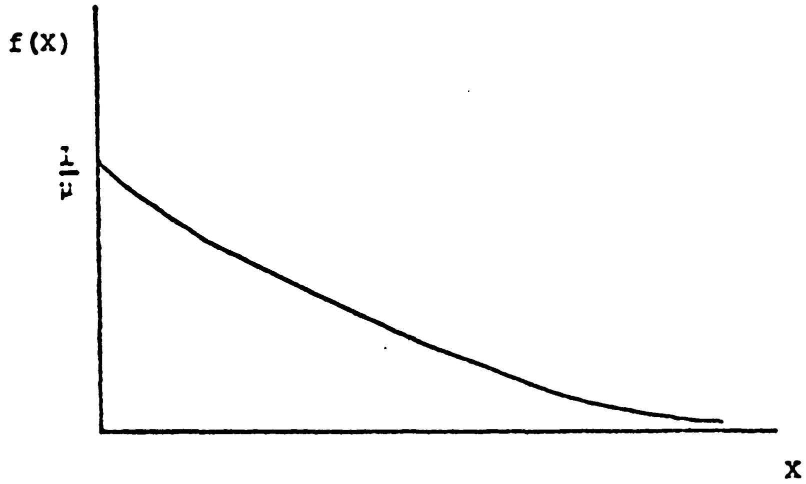
i. $\beta(2,11)$ for $1 \leq X \leq 100$; $E(X) = 15.4$.
ii. $\beta(2,13)$ for $1 \leq X \leq 150$; $E(X) = 20$.
iii. $\beta(2,14)$ for $1 \leq X \leq 200$; $E(X) = 25$.
iv. $\beta(2,11)$ for $1 \leq X \leq 200$; $E(X) = 30.7$.
v. $\beta(2, 8)$ for $1 \leq X \leq 200$; $E(X) = 40$.

Exponential distribution

The exponential distribution with parameter μ is denoted by $E_x(\mu)$ with probability density function $f(X) = \frac{1}{\mu} e^{-X/\mu}$ for $X \geq 0$, where μ is the expected value of variable X. For the purpose of simulating list lengths, X is restricted to the range between 0 and 400 in order to avoid excessively



Beta distribution β(a,b)



Exponential distribution $E_x(\mu)$

Figure 3.8 Beta distribution and exponential distribution long key lists. For each of the five beta distributions in the previous paragraph, a corresponding exponential distribution with similar mean value is used. Hence, five exponential distributions with mean values, μ, equal to 15, 20, 25 30, and 40 are selected. Cumulative probability distributions for the five beta functions and the five exponential functions are listed in Table 3.3. Procedures to generate random variates of the beta distribution and exponential distributions are described in Appendix 1.

The totality of the ten distributions allows one to observe the way in which the performance of the hybrid organizations is affected when

i. minor parameter tunings continually alter the shape of the same distribution, and

ii. the list length distribution changes from a beta function to an exponential function while both have similar mean values.

2. <u>k and R</u>

A value of 10 is used for parameter k, the average number of keys in a record. R is the average number of bytes in a data record; this is assumed equal to 200 bytes in the simulation runs.

X \ distribution	Exponential E(X)=15	Exponential E(X)=20	Exponential E(X)=25	Exponential E(X)=30	Exponential E(X)=40
10	0.482	0.391	0.327	0.278	0.215
20	0.728	0.626	0.544	0.482	0.391
30	0.859	0.764	0.691	0.626	0.523
40	0.922	0.859	0.787	0.728	0.626

Cumulative exponential distribution

X \ distribution	$\beta(2,11)$ $1<X<100$ $E(X)=15.4$	$\beta(2,13)$ $1<X<150$ $E(x)=20$	$\beta(2,14)$ $1<X<200$ $E(X)=25$	$\beta(2,11)$ $1<X<200$ $E(X)=30.7$	$\beta(2,8)$ $1<X<200$ $E(X)=40$
10	0.353	0.255	0.180	0.123	0.074
20	0.721	0.584	0.469	0.353	0.237
30	0.914	0.799	0.686	0.563	0.418
40	0.980	0.922	0.828	0.721	0.572

Cumulative beta distribution

Table 3.3 Some exponential and beta distributions

Querying and updating parameters

1. m and n_i

Given queries of the form, $\bigvee_{i=1}^{m} (k_{i,1} \wedge K_{i,2} \wedge \cdots \wedge K_{i,n_i})$, m is the number of products in the expression and n_i is the number of keys used in the i^{th} product. Values for n_i are again assumed to have a beta distribution since a bell-shaped curve closely represents the likelihood of the number of keys within each product of a user query. In order to experiment with different environments in which the n_i values are typically large or small, five instances of the beta distribution are selected with different parameter settings to control the sizes of n_i, they are :

i. Beta distribution with parameters a = 3, b = 2, and the range of the variable n_i being between 1 and 6, that is, $\beta(3,2)$ for $1 \leq n_i \leq 6$, mean value $E(n_i)$ is equal to 4.12.

ii. $\beta(4,3)$ for $1 \leq n_i \leq 6$, $E(n_i) = 3.95$.

iii. $\beta(3,2)$ for $1 \leq n_i \leq 5$, $E(n_i) = 3.50$.

iv. $\beta(4,2)$ for $1 \leq n_i \leq 4$, $E(n_i) = 3.16$.

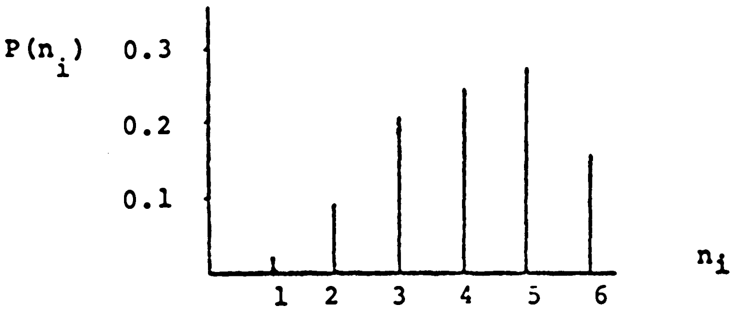
v. $\beta(3,2)$ for $1 \leq n_i \leq 4$, $E(n_i) = 2.9$.

Illustrations of these distributions are shown in Figure 3.9. The data in Fig. 3.9 are collected from Pearson (1968).

The number of products, m, in a query is generally not too large. Values of 1, 2, 3 are assigned to m in this simulation process with equal probabilities.

1) $\beta(3,2)$ for $1 \leq n_i \leq 6$ with $E(n_i) = 4.12$

n_i	1	2	3	4	5	6
$P(n_i)$	0.0144	0.0938	0.2043	0.2682	0.2827	0.1366



2) $\beta(4,3)$ for $1 \leq n_i \leq 6$ with $E(n_i) = 3.95$

n_i	1	2	3	4	5	6
$P(n_i)$	0.0075	0.0893	0.2469	0.3235	0.2672	0.0656

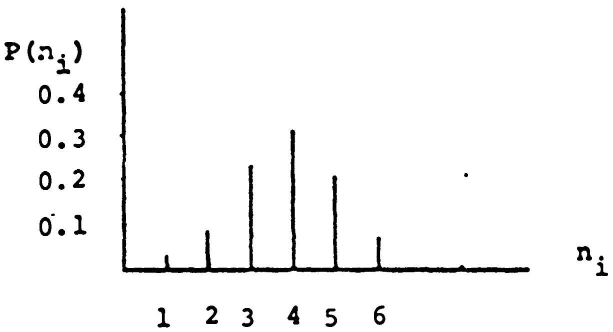
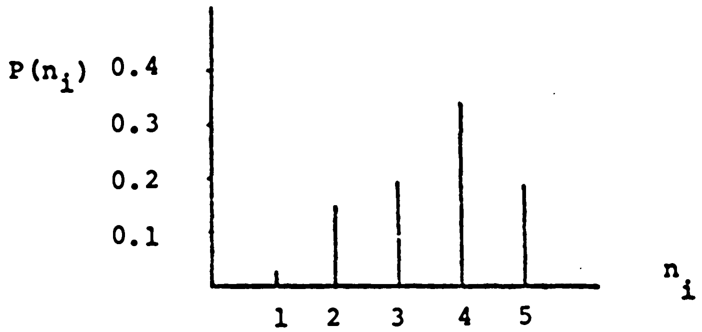


Figure 3.9 Beta distributions for n_i.

3) $\beta(3,2)$ for $1 \leq n_i \leq 5$ with $E(n_i) = 3.5$

n_i	1	2	3	4	5
$P(n_i)$	0.0272	0.1520	0.2960	0.3440	0.1805



4) $\beta(4,2)$ for $1 \leq n_i \leq 4$ with $E(n_i) = 3.16$

n_i	1	2	3	4
$P(n_i)$	0.0156	0.1719	0.4453	0.3672

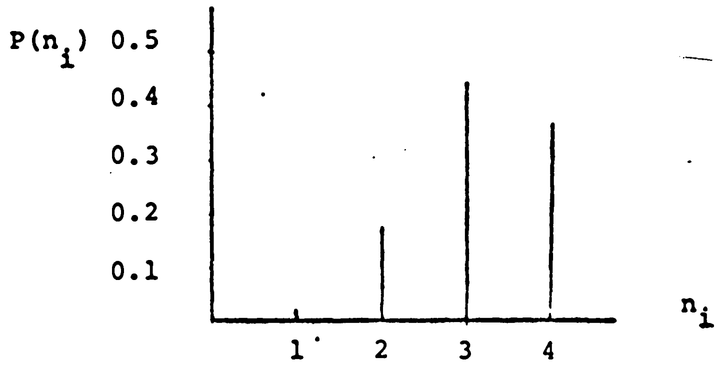


Figure 3.9 (Continued)

5) $\beta(3,2)$ for $1 \leq n_i \leq 4$ with $E(n_i) = 2.9$

n_i	1	2	3	4
$P(n_i)$	0.0508	0.2617	0.4258	0.2617

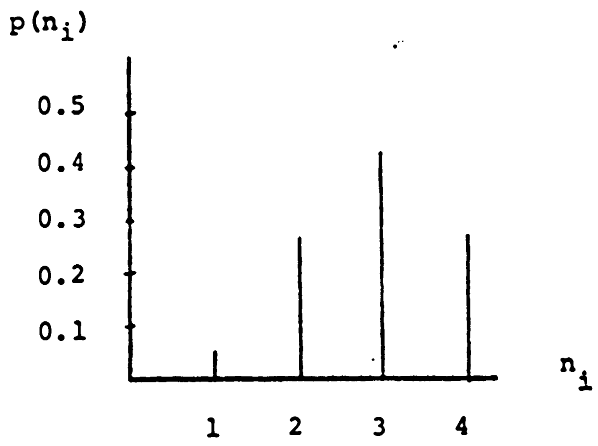


Figure 3.9 (Continued)

2. $\ell_{i,j}$ and ℓ_i

Assume that the distribution of the list lengths of the keys used in the queries is the same as the list length distribution for all the keys existing in the file -- the latter distribution was defined previously to be any one of the ten beta and exponential distributions. This assures that certain anomalous situations do not occur such as the query keys appearing uniformly in very few records while most of the key lists in the file are very long. Hence, for each key $K_{i,j}$ in a query, the corresponding list length $\ell_{i,j}$ is assigned a value by generating a random variate from the distribution function which characterizes the list lengths of all keys in the file.

The next parameter to be determined is ℓ_i, the number of data records satisfying a product $P_i = K_{i,1} \wedge K_{i,2} \wedge \cdots \wedge K_{i,n_i}$ in the query. This part of the simulation is based on two observations :

a) ℓ_i is always smaller than or equal to $\ell_{i,1}$ by definition. (note the convention of $\ell_{i,1} \leq \ell_{i,2} \leq \cdots \leq \ell_{i,n_i}$ for the list lengths of the n_i keys in P_i).

b) The more keys are connected by the logical AND operation ($\wedge$) in P_i, the smaller will be ℓ_i as a fraction of $\ell_{i,1}$.

To implement observation b), the following definition is necessary :

<u>Definition</u> $\rho(1), \rho(2), \cdots, \rho(k)$ is said to be a series of

<u>intersection factors</u> if the condition $1 = \rho(1) \geq \rho(2) \geq \ldots \geq \rho(k)$ is true for any positive integer k.

Incorporating this reasoning, ℓ_i is defined as $\ell_i = \ell_{i,1}\,\rho(n_i)$. The particular values adopted for the intersection factors in this simulation are $\rho(1) = 1$, $\rho(2) = 0.55$, $\rho(3) = 0.4$, $\rho(4) = 0.35$, $\rho(5) = 0.32$ and $\rho(6) = 0.3$. Intersection factors $\rho(k)$ for $k > 6$ are not needed in this simulation since an upper bound of 6 is used for the values of parameter n_i.

The same reasoning can be applied to determine the number of records containing at least one of any two given keys. For example, if $K_{i,1}$ and $K_{i,2}$ appear in $\ell_{i,1}$ and $\ell_{i,2}$ records respectively, then the number of records containing at least one of the two keys is $\ell_{i,1} + \ell_{i,2} - \min\{\ell_{i,1}, \ell_{i,2}\}\,\rho(2)$.

Two additional values $T_i(\ell_{i,1}, \ell_{i,2})$ and $T_u(\ell_{i,1}, \ell_{i,2})$, the time needed to perform a list intersection or a list union operation on sorted list, in CPU, must be estimated. Two IBM 360 assembly language programs were written to perform these tasks and the estimated timings in units of mini-seconds can be expressed as

$$T_i(\ell_{i,1}, \ell_{i,2}) = 0.021 + \ell_\Lambda\, 0.0218 + (\ell_{i,1} + \ell_{i,2} - 2\,\ell_\Lambda)\, 0.0048$$

and

$$T_u(\ell_{i,1},\ell_{i,2}) = 0.021 + (\ell_{i,1} + \ell_{i,2} - \ell_\Lambda)\ 0.0218$$

where $\ell_\Lambda = \min\{\ell_{i,1}, \ell_{i,2}\}\ \rho(2)$ accounts for the intersection part of the two lists.

3.5.2 Simulation Results on Querying Costs

The algorithm to process a query in a hybrid list organization with parameter ℓ_{th} can be summarized as follows :

1) Collect every query product P_i whose shortest list length, $\ell_{i,1}$, is smaller than or equal to ℓ_{th} and process the sum of these products according to the search algorithm for multilists.

2) Process the sum of the remaining products by the algorithm for inverted lists.

Using the model set up in the previous section, this procedure can be easily simulated by the following steps :

Step 1 : (Initialization) $H(\ell_{th}) = 0$ for $\ell_{th} = 0, 2, \cdots, 48, 50$, and ∞, where $H(\ell_{th})$ is the cost of processing a total of NQ queries in a hybrid organization of parameter ℓ_{th}. Denote $H(0)$ by V and $H(\infty)$ by M.

Step 2 : (NQ queries are to be simulated) Repeat steps 3 to 9 for NQ times. (Steps 3 to 6 generate one query,

$$Q = \bigvee_{i=1}^{m} (K_{i,1} \wedge K_{i,2} \wedge \cdots \wedge K_{i,n_i}),$$ and steps 7 to 9 estimate costs of processing such a query in each of the 27 hybrid organizations corresponding

to $\ell_{th} = 0, 2, 4, \cdots 50$ and ∞).

<u>Step 3</u> : Generate m, number of products in Q. Repeat steps 4 to 6 for $i = 1, 2, \cdots, m$.

<u>Step 4</u> : Generate n_i, number of keys in product P_i.

<u>Step 5</u> : Generate $\ell_{i,1}, \ell_{i,2}, \cdots, \ell_{i,n_i}$.

<u>Step 6</u> : Compute $\ell_i = \rho(n_i)\ \ell_{i,1}$.

<u>Step 7</u> : (Compute the cost for hybrid organizations of different ℓ_{th} values). Repeat steps 8 and 9 for $\ell_{th} = 0, 2, 4, \cdots, 48, 50$ and ∞.

<u>Step 8</u> : Collect every product P_i whose shortest list length $\ell_{i,1}$ is smaller than or equal to ℓ_{th}. Assuming that the sum of these products is processed by the search algorithm for multilists, estimate the cost by equation QC_M.

<u>Step 9</u> : The remaining products are to be processed by inverted lists search algorithm. Estimate the cost by equation QC_I and combined with the cost estimated in step 8. Set the combined cots to $H(\ell_{th})$.

<u>END</u>.

Twenty seven elements in the class of hybrid list organizations are tested in the simulation. Values of ℓ_{th} range from 0 to 50 plus infinity. When ℓ_{th} is equal to zero or infinity, degenerate organizations --- inverted lists and multilist organizations respectively --- are produced. A total of 400 queries are generated and the average cost of each organization is computed. As indicated in the previous section, the simulation procedure was repeated for each of the five probability distributions for parameter n_i and each of the ten distributions for parameter $\ell_{i,j}$. The results are described in Table 3.4 to 3.7 and interpreted in the following remarks.

<u>Remark 1</u> The results in Table 3.4 are obtained by assuming that the list length parameter $\ell_{i,j}$ has an exponential distribution of mean value 20, and the values of $\ell_{i,j}$ are in the interval between 1 and 400. (V - H) / V and (M - H) / M, the percentage improvements in querying costs for a hybrid organization over the costs for inverted list and multilist organizations, are displayed at various ℓ_{th} values for each of the five distributions assumed for n_i, the number of keys in a query product P_i. As ℓ_{th} increases from zero to infinity, both (V - H) / V and (M - H) / M climb to positive peak values and then drop gradually. At one end of the ℓ_{th} spectrum, where ℓ_{th} = o, (V - H) / V is zero because the corresponding hybrid organization degenerates into an inverted list organization, and the (M - H) / M value at this point actually amounts to the performance advantage of an inverted list organization over a multilist organization. At the other end of the spectrum where ℓ_{th} is infinity, (M - H) / M becomes

ℓ_{th} \ $E(n_i)$ / P.I.	4.12		3.95		3.50		3.16		2.90	
	$\frac{V-H}{V}$	$\frac{M-H}{M}$	$\frac{V-H}{V}$	$\frac{M-H}{M}$	$\frac{V-H}{V}$	$\frac{M-H}{M}$	$\frac{V-H}{V}$	$\frac{M-H}{M}$	$\frac{V-H}{V}$	$\frac{M-H}{M}$
0	0.0	-21.0	0.0	-19.3	0.0	-7.6	0.0	-3.5	0.0	2.0
2	8.2	-11.0	8.0	-9.7	5.8	-1.3	4.9	1.6	3.9	5.8
4	21.1	4.6	20.4	5.1	15.9	9.5	15.1	12.2	11.8	13.6
6	25.9	10.3	24.4	9.9	19.3	13.2	18.3	15.4	14.1	15.8
8	27.5	12.3	25.6	11.3	20.4	14.4	18.4	15.6	14.7	16.4
10	27.3	12.0	25.3	10.9	20.4	14.3	17.0	14.7	14.3	16.1
12	26.3	10.8	24.2	9.0	19.3	13.2	15.9	12.9	12.0	14.4
14	25.4	9.7	23.1	8.3	18.9	12.6	14.4	11.5	11.1	12.9
16	24.4	8.5	21.9	6.9	16.5	10.1	12.9	9.9	9.2	11.0
18	23.3	7.1	21.0	5.8	15.3	8.9	11.8	8.7	8.2	10.0
20	22.5	6.3	20.3	4.9	14.5	8.1	10.7	7.6	7.0	8.9
22	22.1	5.7	19.6	4.1	13.8	7.3	9.3	6.1	5.5	7.4
24	21.6	5.2	18.9	3.3	13.0	6.4	8.2	5.0	4.5	6.5
26	21.1	4.5	18.7	3.1	12.3	5.6	7.4	4.2	3.0	4.9
28	20.3	3.6	18.1	2.4	11.3	4.6	7.1	3.9	2.2	4.2
30	19.8	2.9	17.7	1.8	11.0	4.2	6.6	3.4	1.4	3.4
32	19.4	2.4	17.7	1.8	10.7	4.0	6.0	2.7	0.7	2.7
34	19.1	2.1	17.3	1.3	10.0	3.2	5.4	2.1	0.5	2.5
36	18.5	1.4	17.0	1.1	9.5	2.6	4.6	1.3	-0.2	1.9
∞	17.4	0.0	16.2	0.0	7.03	0.0	3.4	0.0	-2.0	0.0

Legend : $E(n_i)$ = average number of keys within a product in a query
V = cost of querying in inverted file.
M = cost of querying in multilist.
H = cost of querying in hybrid file with parameter ℓ_{th}.
P.I. = % improvement

<u>Table 3.4</u> Percentage querying cost improvements of hybrid file with parameter ℓ_{th} over inverted file and multilist file.
Parameter $\ell_{i,j}$ has exponential distribution with mean = 20 and $1 \leq \ell_{i,j} \leq 400$.

zero for similar reasons. The improvements of (V - H) / V and (M - H) / M as a function of parameter ℓ_{th} are illustrated in Figure 3.10 where n_i has a $\beta(3,2)$ distribution within the range (1,5) and $\ell_{i,j}$ an exponential distribution with mean value 20.

A very interesting and important result shown in Figure 3.10 is that both (V - H) / V and (M - H) / M reach peak values at approximately the same ℓ_{th} values. Also, the tops of the saddle curves are fairly flat. This implies that one can indeed select a non-degenerate hybrid list organization which is better than both an inverted list and a multilist organization. Furthermore, near optimal values of (V - H) / V and (M - H) / M can be achieved simultaneously by the same of hybrid organization. The flatness of the saddle curves also suggests the existence of many instances of "good" non-degenerate hybrid list organizations.

Such observations are not unique to the special case where parameters $\ell_{i,j}$ and n_i have the above-mentioned distributions. Similar conclusions can also be drawn from Table 3.5 where the beta distribution $\beta(2,13)$ is used for parameter $\ell_{i,j}$ in the range of (1,200). As a matter of fact, the same phenomenon is observed for all the ten distributions adopted for $\ell_{i,j}$.

Hence, there is strong experimental support for the following statement :

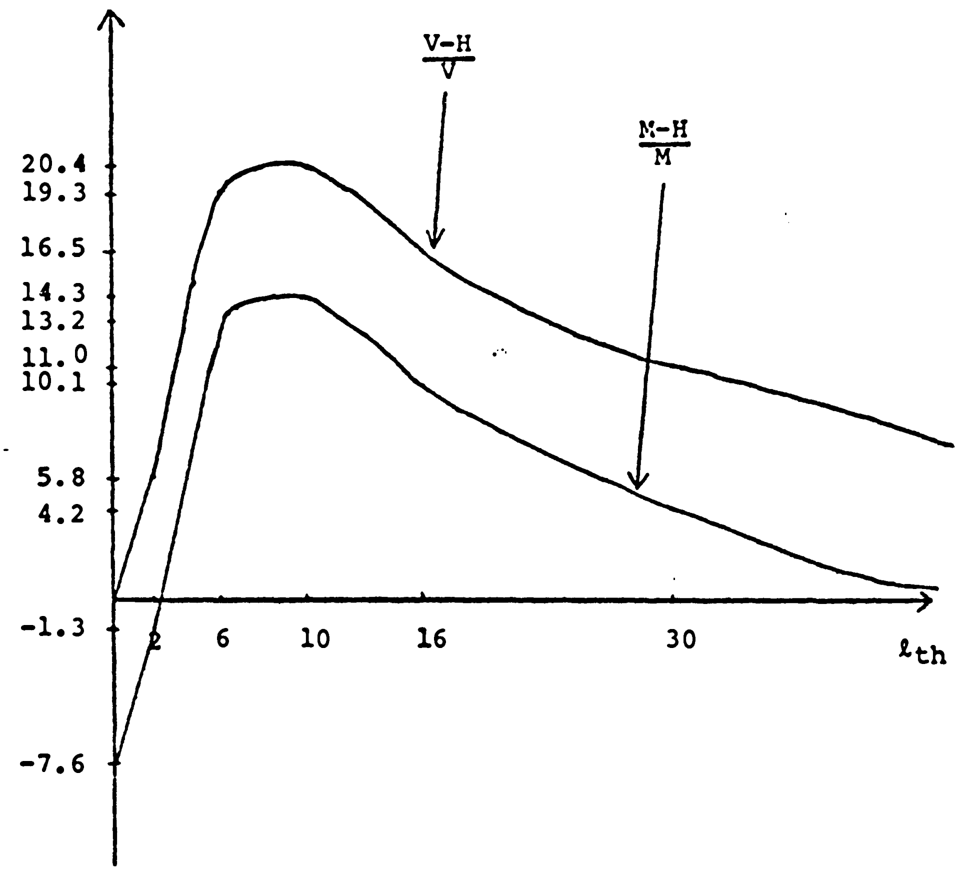


Figure 3.10 Percentage improvement in processing queries for hybrid list organizations versus inverted list and multilist organizations. The distribution of $\ell_{i,j}$ is exponential with mean 20 and the distribution of n_i is beta with mean 3.50.

$E(n_i)$ / P.I. / ℓ_{th}	4.12		3.95		3.50		3.16		2.90	
	$\frac{V-H}{V}$	$\frac{M-H}{M}$	$\frac{V-H}{V}$	$\frac{M-H}{M}$	$\frac{V-H}{V}$	$\frac{M-H}{M}$	$\frac{V-H}{V}$	$\frac{M-H}{M}$	$\frac{V-H}{V}$	$\frac{M-H}{M}$
0	0.0	-5.2	0.0	-1.4	0.0	2.4	0.0	9.16	0.0	9.8
2	2.4	-2.6	2.6	1.2	2.3	4.4	1.8	10.8	1.4	11.1
4	8.0	3.3	8.0	6.7	6.6	8.8	5.6	14.2	4.5	13.9
6	13.7	9.3	13.8	12.6	10.8	12.9	8.9	17.2	7.0	16.1
8	16.2	11.9	15.4	14.2	12.0	14.1	9.7	17.9	7.8	16.9
10	17.0	12.8	15.6	14.4	11.9	14.0	9.0	17.4	7.8	16.9
12	16.2	11.8	14.3	13.1	10.6	12.8	7.0	15.5	6.1	15.3
14	14.3	9.9	12.0	10.8	8.7	10.9	4.6	13.3	3.8	13.2
16	12.8	8.3	10.2	8.9	7.1	9.3	1.6	10.6	2.1	11.7
18	11.3	6.8	8.0	6.7	5.4	7.6	-1.0	8.3	0.2	10.0
20	9.9	5.3	7.0	5.6	4.2	6.4	-2.9	6.5	-1.6	8.4
22	8.7	4.0	5.5	4.1	2.6	4.9	-5.3	4.4	-3.5	6.7
24	7.8	3.0	4.6	3.2	1.0	3.3	-7.0	2.8	-4.9	5.4
26	6.7	1.9	2.9	1.5	-0.4	1.9	-8.1	1.8	-6.6	3.9
28	6.7	1.9	2.5	1.1	-0.7	1.7	-8.6	1.3	-7.6	3.0
30	5.8	0.9	1.0	0.5	-1.4	1.0	-8.9	1.0	-8.0	2.6
32	5.4	0.5	1.9	0.5	-1.6	0.8	-9.5	0.6	-9.4	1.4
34	5.1	0.2	1.4	0.0	-1.8	0.6	-9.9	0.2	-9.8	1.0
36	5.1	0.2	1.4	0.0	-1.9	0.5	-9.9	0.2	-10.1	0.7
∞	4.9	0.0	1.4	0.0	-2.4	0.0	-10.0	0.0	-10.9	0.0

P.I. = % improvement

Table 3.5 Percentage querying cost improvement of hybrid file with parameter ℓ_{th} over inverted file and multi-list file Parameter $\ell_{i,j}$ has beta distribution $\beta(2,13)$ with mean = 20 and $1 \leq \ell_{i,j} \leq 200$.

Among all the file elements in the class of hybrid list organizations, the one which achieves optimal performance in processing queries need not be, and generally is not, the one corresponding to either of the end points in the ℓ_{th} spectrum. Furthermore, nearly optimal performances are achieved for a fair number of file elements within the class.

It is worth to notice that the above conclusion is consistent with the analytic result obtain in lemma 3 of section 3.4 in which the simplified model is used.

Remark 2 Displayed in Table 3.6 and 3.7 are the percentage improvements, (V - H) / V and (M - H) / M, for the optimal hybrid organizations when various $\ell_{i,j}$ and n_i distributions are used. In both tables, it is found that when the lists in a file grow longer, i.e. the distribution of $\ell_{i,j}$ has a larger mean value, the improvement over the inverted file goes down and the improvement over the multilist file goes up. This is consistent with the intuitive notion that an inverted list organization is more suitable for files with long lists while a multilist file is generally the better choice when the lists are short.

Remark 3 A comparison of Table 3.6 and 3.7 reveals that the optimal hybrid list organization achieves better improvements when the list lengths are exponentially distributed than when they are described by beta functions. This can be

file ($\ell_{i,j}$)	P.I. \ $E(n_i)$	4.12	3.95	3.50	3.16	2.90
Exponential $\mu=15$	$\frac{V-H}{V}$	32.2	30.5	24.4	22.3	18.4
	$\frac{M-H}{M}$	8.6	7.9	10.2	11.6	12.5
	ℓ_{th}	8	8	9	8	8
Exponential $\mu=20$	$\frac{V-H}{V}$	27.7	25.6	20.4	18.4	14.7
	$\frac{M-H}{M}$	12.5	11.2	14.3	15.6	16.4
	ℓ_{th}	9	9	10	8	9
Exponential $\mu=25$	$\frac{V-H}{V}$	23.7	22.1	16.9	15.2	11.7
	$\frac{M-H}{M}$	15.2	15.3	17.8	19.5	19.3
	ℓ_{th}	10	9	10	9	9
Exponential $\mu=30$	$\frac{V-H}{V}$	20.8	19.2	14.3	13.0	9.9
	$\frac{M-H}{M}$	17.7	17.5	20.3	22.4	22.1
	ℓ_{th}	11	11	9	9	9
Exponential $\mu=40$	$\frac{V-H}{V}$	16.7	15.9	11.2	10.1	7.3
	$\frac{M-H}{M}$	22.1	22.5	24.5	26.3	.25.6
	ℓ_{th}	11	11	10	10	9

P.I. = % improvement at ℓ_{th}.

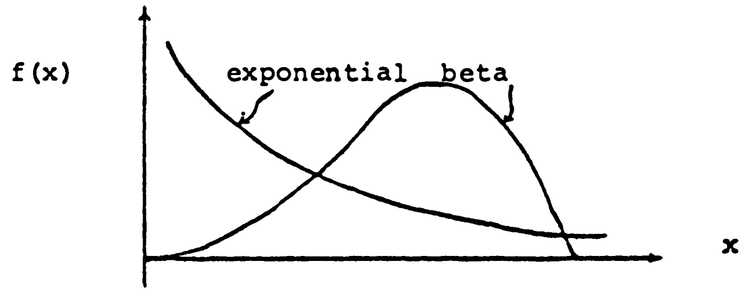
Table 3.6 Querying cost improvements of hybrid file with parameter ℓ_{th} over inverted file and multilist file for five different query characteristics ($E(n_i)$) and five different exponential list lengths characteristics.

file ($\ell_{i,j}$)	$E(n_i)$ / P. I.	4.12	3.95	3.50	3.16	2.90
$\beta(2,11)$ $\mu=15.4$	$\frac{V-H}{V}$	23.3	21.3	16.7	13.5	11.6
	$\frac{M-H}{M}$	8.6	9.5	10.5	13.1	12.4
	ℓ_{th}	9	9	9	8	8
$\beta(2.13)$ $\mu=20$	$\frac{V-H}{V}$	17.0	15.7	12.4	9.7	8.0
	$\frac{M-H}{M}$	12.7	14.5	14.4	18.0	17.0
	ℓ_{th}	10	9	9	9	9
$\beta(2,14)$ $\mu=25$	$\frac{V-H}{V}$	12.5	11.9	8.7	7.0	5.4
	$\frac{M-H}{M}$	16.7	18.1	18.0	20.7	18.7
	ℓ_{th}	10	10	9	8	9
$\beta(2,11)$ $\mu=30.7$	$\frac{V-H}{V}$	9.3	8.1	6.3	4.9	3.8
	$\frac{M-H}{M}$	19.6	21.5	23.2	25.7	23.5
	ℓ_{th}	10	10	10	10	10
$\beta(2,8)$ $\mu=40$	$\frac{V-H}{V}$	6.2	5.9	3.7	3.0	2.2
	$\frac{M-H}{M}$	27.2	28.3	28.4	31.5	28.7
	ℓ_{th}	12	11	12	11	12

P. I. = % improvement at ℓ_{th}.

Table 3.7 Querying cost improvements of hybrid file with parameter ℓ_{th} over inverted file and multilist file for five different query characteristics ($E(n_i)$) and five different beta list lengths characteristics.

explained by the shape of the two types of distribution density curves :



The beta distributions concentrate most of its probability density in the middle of the range of x, while the exponential distribution has larger densities than the beta distribution at the lower and higher ends of x. Note that (a) a multilist is better than an inverted list when the list is short and vice versa; and (b) the hybrid list organization selects a list implementation between an inverted list and a multilist depending on the list length. When one compares the exponential curve and the beta curve, one discovers that the use of exponential distribution for list lengths would create more lists that are very short --- for which the multilist processing is better --- and also more lists that are very long --- for which the inverted list processing is better; on the other hand, the use of beta distribution for list lengths would create more lists whose lengths are in the middle range --- where the performances between multilist processing and inverted list processing are not distinctly different. Hence, hybrid list organizations achieve better improvements when the list lengths are exponentially distributed than when the lists are described by a beta distribution.

3.5.3 Simulation Results on Querying Plus Updating Costs

The costs for the five types of update operations, UC_{Ii} and UC_{Mi} for i = 1, 2, 3, 4, 5, for an inverted list organization and a multilist organization were tabulated in Figure 3.6 of section 3.2.2. Each of the ten equations can be easily evaluated by substituting the parameter values determined in section 3.5.1. To estimate the average cost per update operation, UC_I and UC_M, it is necessary to determine values for P_1, P_2, P_3, P_4, and P_5, the probabilities for the five types of updates. Given the values for P_i's, UC_I and UC_M can be written as $UC_I = \sum_{i=1}^{5} UC_{Ii} P_i$ and $UC_M = \sum_{i=1}^{5} UC_{Mi} P_i$. The values of P_i's may be very different in each individual system. In this simulation, P_1 and P_2 are assumed to be 0.25, P_3 is equal to 0.4 and both of P_4 and P_5 are 0.05.

The file maintenance procedures in a non-degenerate hybrid organization depend on whether the keys involved in the updates are stored as inverted list or multilist structures. Consider updates of type one and two where a key is added to or deleted from a data record. If the corresponding key list is stored as an inverted list then the update procedures of inverted list structure are followed; otherwise one uses multilist update procedures. Type three updates are independent of the particualr list structure since they modify non-key data only. For type four and type five updates, an average of k keys contained in the deleted or inserted record are to be modified, and the modification of each of the k key lists is equivalent to a type one or type two operation. Hence, it is necessary to determine which list structure was used for the keys for type one or type two updates. Similarly, in a type four or type five update, one should know how many of the k keys were stored as inverted lists and multilists respectively.

Both questions can be answered easily if one adopts the reasonable assumption that when a proportion $P_I (P_M)$ of the keys is stored as inverted lists (multilists), then a fraction $P_I (P_M)$ of the keys involved in an update operations will also consist of inverted lists (multilists). The average cost of an update operation in a hybrid organization is then $UC_H = P_I \, UC_I + P_M \, UC_M$, where $P_I + P_M$ is of course equal to one.

P_I and P_M are readily available when the key length distribution is known. In section 3.5.1, five versions of exponential distribution and five versions of beta distribution were used for the descriptions of key length parameter X.

Given any one of the ten distributions, say, exponential distribution with mean value 20 with probability density function $f(x) = \frac{1}{20} e^{-x/20}$ for $x > 0$, P_I and P_M for a hybrid organization of parameter ℓ_{th} are $P_M = \int_0^{\ell_{th}} f(x)\, dx$ and $P_I = 1 - P_M$.

In summary, the average costs per update operation in a pure inverted list organization, or pure multilist organization, or non-degenerate hybrid list organization can be estimated by the following equations :

$$UC_I = \sum_{i=1}^{5} P_i\, UC_{Ii},$$

$$UC_M = \sum_{i=1}^{5} P_i\, UC_{Mi},$$

and

$$UC_H = P_I\, UC_I + P_M\, UC_M.$$

The total costs including querying and updating are computed as

$$C_I = P_Q\, QC_I + P_U\, UC_I,$$

$$C_M = P_Q\, QC_M + P_U\, UC_M,$$

and

$$C_H = P_Q\, QC_H + P_u\, UC_H$$

where P_Q and P_M are the proportion of querying and updating requests processed in a system. The simulation results for C_I , C_M and C_H are shown in Table 3.8 to 3.13.

In Table 3.8 and 3.9, the list length distributions are exponential and ninety or eighty percent of the transections

file ($\ell_{i,j}$)	$E(n_i)$ / P.I.	4.12	3.95	3.50	3.16	2.90
Exponential $\mu=15$	$\frac{V-H}{V}$	31.8	30.2	24.5	22.3	18.6
	$\frac{M-H}{M}$	6.3	5.6	8.4	9.4	10.5
	ℓ_{th}	8	9	8	8	8
Exponential $\mu=20$	$\frac{V-H}{V}$	27.4	25.3	20.5	18.3	14.9
	$\frac{M-H}{M}$	10.5	9.2	12.7	13.4	14.7
	ℓ_{th}	9	9	9	9	9
Exponential $\mu=25$	$\frac{V-H}{V}$	23.5	21.7	16.9	15.1	11.8
	$\frac{M-H}{M}$	13.4	13.1	16.1	17.6	17.7
	ℓ_{th}	10	10	10	9	9
Exponential $\mu=30$	$\frac{V-H}{V}$	20.6	19.1	14.1	13.1	10.0
	$\frac{M-H}{M}$	16.0	15.8	18.5	20.7	20.6
	ℓ_{th}	11	11	10	9	9
Exponential $\mu=40$	$\frac{V-H}{V}$	16.6	15.9	11.2	10.2	7.4
	$\frac{M-H}{M}$	20.6	20.9	23.1	24.9	24.3
	ℓ_{th}	11	11	10	10	9

P.I. = % improvement at ℓ_{th}.

Table 3.8 Cost improvements of hybrid file with parameter ℓ_{th} over inverted & multilist files for exponential list lengths distributions. 90% of the transactions are queries, and 10% of the transactions are updates.

file $(\ell_{i,j})$ \ P.I. \ $E(n_i)$		4.12	3.95	3.50	3.16	2.90
Exponential $\mu=15$	$\frac{V-H}{V}$	31.3	29.9	24.4	22.3	18.8
	$\frac{M-H}{M}$	3.6	3.1	5.8	6.8	8.1
	ℓ_{th}	8	10	9	8	8
Exponential $\mu=20$	$\frac{V-H}{V}$	27.0	25.1	20.5	18.3	15.1
	$\frac{M-H}{M}$	8.1	6.8	10.5	11.3	12.3
	ℓ_{th}	9	9	10	8	9
Exponential $\mu=25$	$\frac{V-H}{V}$	23.2	21.7	16.9	15.1	12.0
	$\frac{M-H}{M}$	11.2	11.0	14.1	15.4	15.8
	ℓ_{th}	11	9	10	9	9
Exponential $\mu=30$	$\frac{V-H}{V}$	20.4	19.1	14.2	13.0	10.1
	$\frac{M-H}{M}$	14.0	13.8	16.7	18.7	18.9
	ℓ_{th}	11	10	9	10	10
Exponential $\mu=40$	$\frac{V-H}{V}$	16.5	15.8	11.2	10.2	7.5
	$\frac{M-H}{M}$	18.7	19.0	21.4	23.0	22.7
	ℓ_{th}	11	11	10	11	11

P.I. = % improvementsat ℓ_{th}.

Table 3.9 Cost improvements of hybrid file with parameter ℓ_{th} over inverted and multi-list files for exponential list lengths distributions. 80% of the transactions are queries and 20% of the transactions are updates.

are assumed to consists of query processing. Table 3.10 and 3.11 display similar results when beta distributions are used to describe the list lengths. All four tables show phenomena resembling those derived from the previous section where $P_u = 0$ and $P_Q = 1$.

To indicate how varying portions of queries and updates affect the improvements of hybrid list organizations, Table 3.12 and 3.13 exhibits improvements (V - H) / V and (M - H) / M at various P_Q and P_U values. It can been seen that when the proportion of update requests increases, the improvement of the nondegenerate hybrid organizations over the inverted list organizations changes slightly, while improvements over the multilist organization drop heavily. When P_Q is equal to 0.5 where fifty percent of the requests are update operations, no non-degenerate hybrid list organizations can do better than a pure multilist organization. This phenomenon is understandable because unsorted multilists are much easier to update than inverted lists as pointed out in section 3.2.2.

In the description of the update procedures for hybrid list organizatios, it is assumed that the list structure of a key does not change because of the updating : That is, if the key list was an inverted list or a multilist before an update operation, it remains so after the update is performed. Such an assumption may disturb the hybrid organization charac-

file $(\ell i,j)$ \ P.I.	E(n_i)	4.12	3.95	3.50	3.16	2.90
$\beta(2,11)$ $\mu=15.4$	$\frac{V-H}{V}$	23.1	21.2	16.9	13.6	11.8
	$\frac{M-H}{M}$	6.5	7.4	8.6	11.1	10.5
	ℓ_{th}	9	9	9	8	8
$\beta(2,13)$ $\mu=20$	$\frac{V-H}{V}$	17.0	15.6	12.4	9.9	8.1
	$\frac{M-H}{M}$	10.9	12.6	12.6	16.1	15.3
	ℓ_{th}	10	9	9	9	9
$\beta(2,14)$ $\mu=25$	$\frac{V-H}{V}$	12.5	11.9	8.7	7.0	5.5
	$\frac{M-H}{M}$	15.1	16.4	16.4	19.0	17.2
	ℓ_{th}	10	10	10	8	9
$\beta(2,11)$ $\mu=30.7$	$\frac{V-H}{V}$	9.2	8.1	6.3	5.0	3.9
	$\frac{M-H}{M}$	18.0	20.0	21.7	24.3	22.2
	ℓ_{th}	11	10	11	10	10
$\beta(2,8)$ $\mu=40$	$\frac{V-H}{V}$	6.2	5.9	3.8	3.1	2.2
	$\frac{M-H}{M}$	25.9	27.0	27.2	30.4	27.6
	ℓ_{th}	12	11	12	11	10

P.I. = % improvement at ℓ_{th}.

Table 3.10 Cost improvements of hybrid file with parameter ℓ_{th} over inverted and multilist file for beta list lengths distributions. 90% of the transactions are queries and 10% of the transactions are updates.

file ($\ell_{i,j}$)	P.I. \ $E(n_i)$	4.12	3.95	3.50	3.16	2.90
$\beta(2,11)$ $\mu=15.4$	$\frac{V-H}{V}$	22.9	21.0	16.9	13.7	11.9
	$\frac{M-H}{M}$	4.1	5.0	6.3	8.7	8.2
	ℓ_{th}	8	9	9	7	8
$\beta(2,13)$ $\mu=20$	$\frac{V-H}{V}$	16.9	15.6	12.4	10.0	8.4
	$\frac{M-H}{M}$	8.6	10.3	10.3	13.9	13.3
	ℓ_{th}	10	10	9	9	10
$\beta(2,14)$ $\mu=25$	$\frac{V-H}{V}$	12.4	11.8	8.8	7.0	5.6
	$\frac{M-H}{M}$	13.1	14.3	14.5	17.0	15.4
	ℓ_{th}	10	11	10	9	9
$\beta(2,11)$ $\mu=30.7$	$\frac{V-H}{V}$	9.2	8.0	6.4	5.0	4.0
	$\frac{M-H}{M}$	16.2	18.1	20.0	22.6	20.5
	ℓ_{th}	11	11	11	10	10
$\beta(2,8)$ $\mu=40$	$\frac{V-H}{V}$	6.2	5.9	3.8	3.1	2.3
	$\frac{M-H}{M}$	24.4	25.5	25.7	28.8	26.3
	ℓ_{th}	12	12	12	12	12

P.I. = % improvement at ℓ_{th}.

Table 3.11 Cost improvement of hybrid file with parameter ℓ_{th} over inverted and multilist file for beta distribution of list lengths. 80% of the transactions are queries and 20% of the transactions are updates.

P_u \ $E(n_i)$	P.I.	4.12	3.95	3.50	3.16	2.90
1.0	$\frac{V-H}{V}$	27.7	25.6	20.6	18.2	14.7
	$\frac{M-H}{M}$	12.5	11.2	14.6	15.3	16.4
0.9	$\frac{V-H}{V}$	27.4	25.3	20.5	18.2	14.9
	$\frac{M-H}{M}$	10.5	9.2	12.7	13.4	14.7
0.8	$\frac{V-H}{V}$	27.0	25.1	20.5	18.3	15.1
	$\frac{M-H}{M}$	8.1	6.8	10.5	11.2	12.3
0.7	$\frac{V-H}{V}$	26.6	24.8	20.4	18.3	15.3
	$\frac{M-H}{M}$	5.2	3.9	7.8	8.4	10.2
0.6	$\frac{V-H}{V}$	26.2	24.4	20.3	18.4	15.5
	$\frac{M-H}{M}$	1.5	0.2	4.4	4.9	7.0
0.5	$\frac{V-H}{V}$	25.6	24.0	20.3	18.5	15.9
	$\frac{M-H}{M}$	-3.2	-4.6	-0.1	0.4	2.8
0.4	$\frac{V-H}{V}$	24.9	23.4	20.2	18.6	16.3
	$\frac{M-H}{M}$	-9.4	-10.8	-6.0	-5.7	-2.8
0.3	$\frac{V-H}{V}$	24.0	22.8	20.0	18.7	16.8
	$\frac{M-H}{M}$	-18.2	-19.6	-14.5	-14.4	-11.0

P.I. = % improvement

Table 3.12 Improvement of hybrid file with $\ell_{th} = 9$ over inverted file and multilist file when P_u porportions of hte transactions are update requests. The list lengths have Exponential distribution with mean 20.

P_u	P.I. \ $E(n_i)$	4.12	3.95	3.50	3.16	2.90
1.0	$\frac{V-H}{V}$	16.7	15.7	12.4	9.7	8.0
	$\frac{M-H}{M}$	12.4	14.5	14.4	18.0	17.0
0.9	$\frac{V-H}{V}$	16.6	15.6	12.4	9.9	8.1
	$\frac{M-H}{M}$	10.5	12.6	12.6	16.1	15.3
0.8	$\frac{V-H}{V}$	16.4	15.6	12.4	10.0	8.4
	$\frac{M-H}{M}$	8.1	10.3	10.3	13.9	13.3
0.7	$\frac{V-H}{V}$	16.3	15.4	12.5	10.2	8.6
	$\frac{M-H}{M}$	5.2	7.3	7.5	11.1	10.7
0.6	$\frac{V-H}{V}$	16.1	15.2	12.5	10.4	8.8
	$\frac{M-H}{M}$	1.5	3.6	3.9	7.5	7.4
0.5	$\frac{V-H}{V}$	15.8	15.0	12.5	10.6	9.2
	$\frac{M-H}{M}$	-3.3	-1.2	-0.7	2.9	3.1
0.4	$\frac{V-H}{V}$	15.5	14.8	12.6	10.9	9.6
	$\frac{M-H}{M}$	-9.8	-7.7	-7.0	-3.4	-2.8
0.3	$\frac{V-H}{V}$	15.1	14.5	12.7	11.2	10.1
	$\frac{M-H}{M}$	-19.0	-17.0	-16.1	-12.5	-11.3

P.I. = % improvement.

Table 3.13 Improvements of hybrid file with $\ell_{th} = 9$ over inverted file and multilist file when P_u porportion of the transactions are update requests. The list lengths are between 1 and 200 and have $\beta(2,13)$ Beta distribution with mean = 20.

terized by parameter ℓ_{th} because it is possible that, after a number of updates, a short list may grow longer than the threshold length ℓ_{th} or an originally long list is shortened below ℓ_{th} . A question of immediate concern is then whether such list length alterations seriously degrade system performance. This answer is probably no. A result in the previous section shows that there exists a reasonably large range of ℓ_{th} values whose corresponding hybrid organizations all have near optimal performances. This implies that minor violations of the list structure constraints in hybrid organizations need not have serious effects on performance. Furthermore, the number of times in which a specific key is deleted from the data records and the number of times in which the same key is added to some other records may well balance out in the course of updating. Hence, the average list lengths may not change dramatically. In most information systems, file reorganizations are regularly scheduled for the purposes of garbage collection and record address reallocation. The lists in a hybrid file organization may then be restructured at reorganization time.

3.6 Summary

Querying and updating procedures for an inverted list organization and a multilist organization are analyzed in section 3.1 and 3.2. An algorithm is given in section 3.1.2 for retrieving exactly once all the records in the union of more than one unsorted multilist. Querying costs using multilists and inverted lists are compared in section 3.3; this serves as a motivation for a new class of hybrid list organizations in which pure multilist and pure inverted list organizations are included as two special elements. Section 3.3 uses a simplified model to show analytically that one can indeed select some instances of hybrid list organizations that are superior to either of inverted list or multilist organizations. Extensive simulation results on a realistic model are obtained in Section 3.5. The simulation results in conclusions consistent with those obtained in a simplified analytical model.

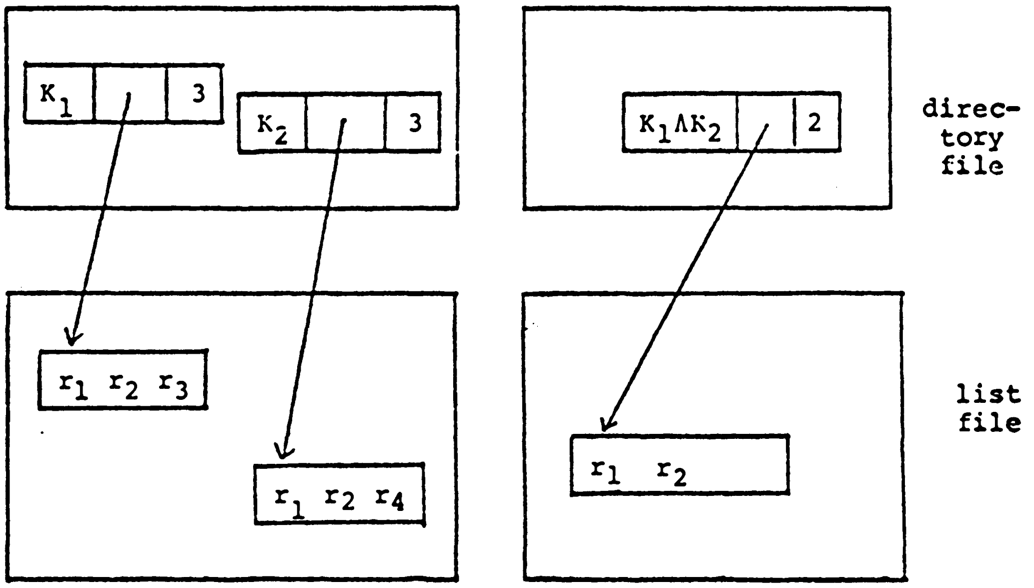
Chapter 4 Combined Indices in List Structure File Organizations

In the previous chapter one aspect of list structure organizations was investigated and the traditional multilist - inverted list dichotomy was extended to a whole class of list organizations. In the present chapter, a different aspect of list structure organizations is discussed with emphasis on the number of keys to be used in list construction. List structure organizations are based on the notion of creating linkages on a set of data records --- either by the use of an inverted list or by the use of a multilist. The records included in a given list are characterized by a specific key attached to each of the records. But the criterion of linking records need not necessarily be restricted to the inclusion of a single key only. If the joint existence of two or more keys is frequently specified by the queries, it is probably advisable to create combined lists such that each list links together a set of data records containing several keys simultaneously. In the case of inverted list organizations, consider the requests for data records containing both key K_1 and key K_2. The existence of an inverted list for such records can reduce the number of list accesses from two to one and can also avoid the list intersection operation. As to multilist organizations, the same request $K_1 \wedge K_2$ does not need to retrieve any data record containing one key but not the other. Figure 4.1 illustrates such an example.

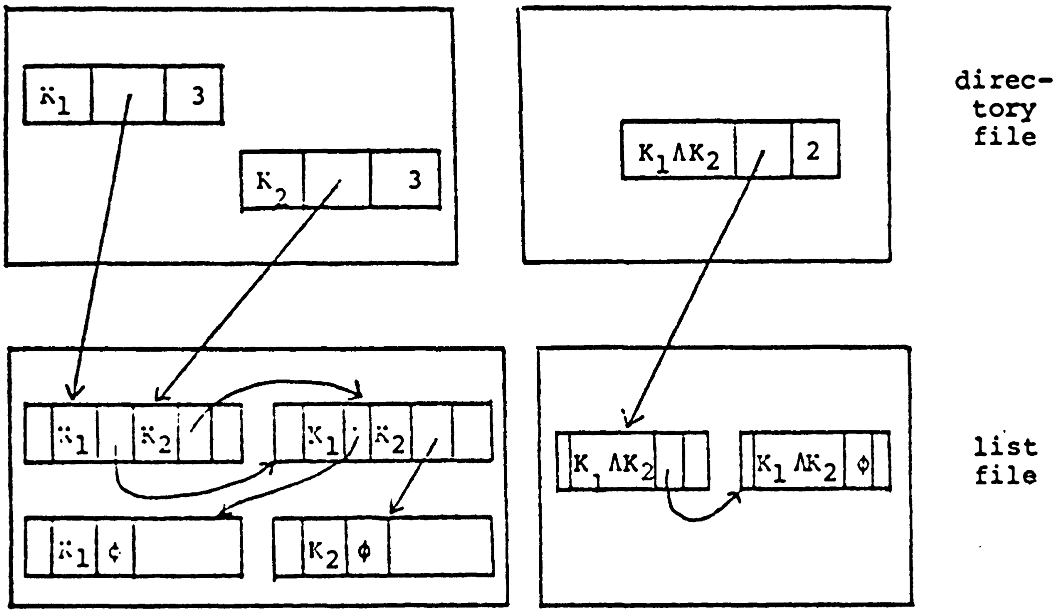
In this chapter, the problem of constructing combined lists where each list contains data records satisfying a criterion based on more than one key is referred to as the Combined Indices Problem. The index to a combined list is called a combined index; otherwise it is called a simple index. The use of combined indices may be attractive for several reasons, but practical and efficient implementations of such ideas must be carefully analyzed. It is the purpose of this chapter to review some previous approaches to this problem and to propose some heuristic solutions.

4.1 Approaches to the Combined Indices Problem

Lum (1970) suggested an approach to the combined index problem for inverted list organization in an attribute based information system. He assumed that the number of attributes appearing in a query product ($K_{i,1} \wedge K_{i,2} \wedge \cdots \wedge K_{i,n_i}$) is at most n. Suppose there are a total of $M (\geq n)$ attributes in the data base and each attribute A_i can have m_i different values $V_{i,1}, V_{i,2}, \cdots, V_{i,m_i}$. A compound index file is constructed for each possible subset of n attributes. Thus there are a total of $\binom{M}{n}$ such compound index files. Each file is organized into buckets such that the record identifiers in each bucket will be records which satisfy an n-tuple of attri-



inverted list organization for a combined index for $K_1 \wedge K_2$.
(data file not shown)



multilist organization for a combined index $K_1 \wedge K_2$.

Figure 4.1 Combined index.

bute-value pairs, one from each attribute. Lum gave a procedure for constructing a compound index file for n attributes, A_1, A_2, $\cdots$, and A_n : The order of concatenation is fixed for a given index file, say, A_1, A_2, $\cdots$, A_n in this example. The location of buckets is assigned by

<u>Step 1</u> : Holding values of attributes A_1, A_2, ..., A_{n-1} fixed and letting attribute A_n take on all its possible values in an increasing sequence.

<u>Step 2</u> : Fixing the first n - 2 attribute values, letting attribute A_{n-1} take its next larger value if any and repeat step 1.

<u>Step 3</u> : When attribute A_{n-1} values have all been accounted for, attribute A_{n-2} takes on its next higher value.

The process is then repeated until all possible n-tuples of the attribute-value pairs are assigned bucket numbers. Figure 4.2 illustrates such an index file. Note that in the actual layout of the compound index file only the third column of the table in Figure 4.2 is stored.

Because of the sequence imposed on the n-tuples in a compound index file, it is possible to compute the bucket address for any n-tuple of attribute-value pairs directly provided that each bucket is of the same size. If a query specifies values for all n attributes A_1, A_2, $\cdots$, A_n, then the identifiers of all data records satisfying this query are contained in a single bucket. If on the other hand a query assigns values to attribute A_1, A_2, $\cdots$, A_k only, then

Bucket numbers	n-tuple attribute-value pairs	record identifier lists
1	$V_{1,1}\ V_{2,1} \cdots V_{n-1,1} \quad V_{n,1}$	$r_{1,1} \quad r_{1,2}, \cdots\cdots$
2	$V_{1,1}\ V_{2,1} \cdots V_{n-1,1} \quad V_{n,2}$	$r_{2,1} \quad r_{2,2}, \cdots\cdots$
	$\vdots$	$\vdots$
m_n	$V_{1,1}\ V_{2,1} \cdots V_{n-1,1} \quad V_{n,m_n}$	$r_{m_n,1} \quad r_{m_n,2}, \cdots\cdots$
m	$V_{1,1}\ V_{2,1} \cdots V_{n-1,2} \quad V_{n,1}$	$r_{m_n+1,1}\, r_{m_n+1,2}, \cdots$
	$\vdots$	$\vdots$
$m_1 m_2 \cdots m_n$	$V_{1,m_1} V_{2,m_2} \cdots V_{n-1,m_{n-1}} V_{n,m_n}$	$\vdots$

The bucket number B of an n-tuple $V_{1,a_1}\ V_{2,a_2} \cdots V_{n,a_n}$ is $B = (a_1 - 1)\, m_2\, m_3 \cdots m_n + (a_2 - 1)\, m_3\, m_4 \cdots m_n$

$+ \cdots\cdots + (a_n - 1)$

Figure 4.2 A compound index file for $A_1, A_2, \cdots, A_n$ in that order.

$m_{k+1}\ m_{k+2} \cdots m_n$ buckets must be accessed. Fortunately, all these buckets are located in adjacent positions in the address space and can be accessed relatively easily.

Problem arises if a compound index file adopts the ordering of $A_1, A_2, \cdots, A_n$ for the n attributes but the $k(<n)$ attributes specified by a query are not the left-most k items $A_1, A_2, \cdots, A_k$. In such situations, many buckets must be accessed in scattered locations. To solve this problem, Lum suggested the use of more than one copy of a compound index file, each copy corresponds to a different permutation of the same n attributes. Lum proved that for any query, one can find a compound index file in which all qualified record-identifier-lists for this query are stored in consecutive buckets if and only if there are $\binom{n}{s}$ copies of a compound index file for any n attributes where each copy represents a different permutation of the n attributes and s is equal to the ceiling of n/2. Suppose that n, the maximum number of attributes used in a query product, is equal to 4 and s is 2. Six copies of the compound index file for attributes A_1, A_2, A_3 and A_4 have to be stored. Each copy corresponds to one of the following attribute permutations : $A_1A_2A_3A_4$, $A_2A_3A_4A_1$, $A_3A_4A_1A_2$, $A_4A_1A_2A_3$, $A_1A_3A_2A_4$, $A_2A_4A_1A_3$.

This method looks very attractive for querying purpose. But the practicality of any file organization is also conditioned upon its balanced performance in file maintenance and upon its storage requirement within realistic limitations. There are indications that Lum's method is weak in these two aspects :

1. The price paid in terms of storage redundancy is very high. Assume that each data record in a file is described by M attributes. In a simple inverted list organization, each record identifier will appear in M inverted lists. However, when Lum's method is applied, every record identifier appears in $\binom{M}{n}$ compound index files, each of which corresponds to a distinct set of n attributes. Furthermore, each one of the $\binom{M}{n}$ compound index files has to be repeated $\binom{n}{s}$ times due to the permuted orderings of the n attributes. Therefore, each record identifier will recur for a total of $\binom{M}{n}\binom{n}{s}$ times. In fact, the ratio of $\binom{M}{n}\binom{n}{s} / M$ serves as a measure of the amount of storage required by Lum's method in comparison with the space needed in a simple inverted list organization. Consider a reasonable data base in which M is equal to 10 and n is 6. Then the ratio is equal to $\binom{10}{6}\binom{6}{3} / 10$, which equals 420. If a simple inverted list organization requires one cylinder on an IBM 2314 disk pack for storage purpose, Lum's compound index files will require 420 cylinders, amounting to more than two disk packs. For a smaller value of M, say 9, the ratio is still 187.
2. As illustrated in Figure 4.2, the bucket number B in a compound index file for an n-tuple $V_{1,a_1} V_{2,a_2} \cdots V_{n,a_n}$

is $B = (a_1 - 1)\, m_2\, m_3 \cdots m_n + (a_2 - 1)\, m_3\, m_4 \cdots m_n + + (a_n - 1)$. If every bucket is of the same size Z, then the bucket address of the tuple is $B \cdot Z + b$ assuming that b is the beginning address of the file. Unfortunately, such a fast direct address calculation is possible only at the expense of increased waste in storage space, because Z must be large enough to hold the longest list of record identifiers. Since the distribution of the number of record identifiers in each bucket may not be uniform, a bucket size Z satisfying the above condition would imply a large amount of empty space.

Summarizing these two observations, it seems that the space overhead in Lum's scheme may be prohibitively high for most practical applications.

3. A third observation concerns the file maintenance efficiency of the compound index files. Lum's scheme may render on-line updating impossible. Suppose a record r is described by M attributes $A_1, A_2, \cdots, A_M$ and is characterized by the M-tuple $V_{1,a_1}\, V_{2,a_2} \cdots V_{M,a_M}$. Consider an update operation which changes value V_{1,a_1} of record r to $V_{1,\bar{a}_1}$, then in the combined index file for attributes $A_1, A_2, \cdots, A_n$, record r must be removed from the bucket representing n-tuple $V_{1,a_1}\, V_{2,a_2} \cdots V_{n,a_n}$ to another bucket which represents n-tuple $V_{1,\bar{a}_1}\, V_{2,a_2} \cdots V_{n,a_n}$. Since record r appears in $\binom{M-1}{n-1}\binom{n}{s}$ compound index files containing attribute

A_1, this modification must be performed in all of the $\binom{M-1}{n-1}\cdot\binom{n}{s}$ files. In contrast, in a simple inverted list organization, record r only has to be removed from the list for key (A_1, V_{1,a_1}), and inserted into the list for key (A_1, $V_{1,\bar{a}_1}$). The other problem arising is the process of modifying the $\binom{M-1}{n-1}\binom{n}{s}$ files deals with the consistency of data. If these files are modified one by one, and retrieval operations for other users are allowed to proceed at the same time, then it is possible that some queries will access those files which have already been updated and some will access those which still remain to be modified. Such inconsistencies can cause erroneous retrieval results and are deemed highly undesirable in an on-line information system. If, on the other hand, locking procedures are applied to lock out a portion or the entirety of each of the $\binom{M-1}{n-1}\cdot\binom{n}{s}$ index files, the system overhead may be unacceptable.

4. Finally, it may not be necessary exhaustively to create all possible combined indices. Many of the keys would rarely or never coexist in queries. A more selective approach may then be more practical even though it is less attractive theoretically.

Wong and Chiang (1971) proposed a scheme for the combined index problem based on theories in Boolean algebra. Assuming that C is a class of finite sets, C is called a Boolean algebra if it is closed under union, intersection, and complementation operations. If L is an arbitrary class of finite sets, there exists a smallest Boolean algebra for L which is called the <u>Boolean algebra generated by L</u>, denoted by B(L).

In a data base containing n keys $K_1, K_2, \cdots, K_n$, let $L(K_i)$ be the set of records (or record identifiers) which are characterized by the existence of key K_i. Let the class of finite sets $\{L(K_i), i = 1, 2, \cdots, n\}$ be denoted by L. From the point of view of the file organizations, L is exactly the collections of inverted lists or multilists, for records including K_i. Since a query is essentially a Boolean expression in terms of the $L(K_i)$'s, the Boolean algebra generated by L, B(L), is exactly the collection of all possible sets of records that can be requested by the queries. In these circumstances, the retrieval problem is reduced to the generation of the elements of B(L) in response to any given query. Wong and Chiang proved that there exists a subcollection of disjoint "atomic sets" in B(L) such that any element of B(L) could be expressed as a union of some of these atomic sets. The theorem is reproduced below :

<u>Theorem</u> Let B(L) be the Boolean algebra generated by L = { $L(K_i)$, i = 1, 2, $\cdots$, n}. Let $C_1, C_2, \cdots, C_{2n}$ be the 2^n atomic sets of the form $\bigwedge_{i=1}^{n} \tilde{L}(K_i)$, where $\tilde{L}(K_i) = L(K_i)$ or $\bar{L}(K_i)$, the complement of $L(K_i)$. Then :

(a) C_j and C_k are disjoint if $j \neq k$.

(b) $b \in B(L)$ implies $b \cap C_j = \phi$ or $B \cap C_j = C_j$ for all j.

(c) Every $b \in B(L)$ is a union of some of the C_j's.

The number of possible atomic sets, 2^n, is indeed very large. Although many of them are empty, the number of non-empty ones may still be much larger than the number of keys in the data base. The significance of this theorem is two-fold as summarized by Wong and Chiang :

(a) Because all atomic sets are disjoint, every record is associated with one and only one atomic set.

(b) Every set of records to be retrieved is a union of some atomic sets. In the process of query retrieval, it is never necessary to take intersections, nor does one need to eliminate duplications in taking unions.

With this setting, each query should be expressed in terms of a collection of atomic sets and data records are retrieved only if they are included in any of the nonempty atomic sets. The former is very easy to do and can be best explained by a simple example. Assume that there are only three keys K_1, K_2, K_3 in the data base. Given a query $(K_1 \wedge K_2) \vee (K_1 \wedge K_3)$ in disjunctive normal form, it can be "developed" into a sum of atomic sets of the equivalent expression: $(K_1 \wedge K_2$ $K_3) \vee (K_1 \wedge K_2 \wedge \overline{K}_3) \vee (K_1 \wedge K_2 \wedge K_3) \vee (K_1 \wedge \overline{K}_2 \wedge K_3)$; the duplicated copy of $(K_1 \wedge K_2 \wedge K_3)$ can then be deleted.

This procedure for obtaining a specific atomic set can be implemented by either of the following two methods :

(1) <u>Direct address calculation</u>

Given n keys, one can number the keys in a specific ordering, say K_1, K_2, ..., K_n. Each atomic set ($\tilde{L}(K_1)\cdot \tilde{L}(K_2) \cdots \tilde{L}(K_n)$) can then be considered as a binary number of n bits b_1, b_2, .., b_n such that $b_i = 1$ if $\tilde{L}(K_i) = L(K_i)$ and $b_i = 0$ if $\tilde{L}(K_i) = \overline{L}(K_i)$. Thus, if each atomic set is assigned a bucket of fixed size Z, the binary number b_1, b_2, ..., b_n multiplied by Z can then directly represent the address of the corresponding atomic set. Such a method avoids the use of directories but causes a good deal of wasted storage because the number of atomic sets may be as large as 2^n, while many of them are empty.

(2) Directory look-up

The directory should include one entry for each nonempty atomic set. For a simple inverted list organization or multilist organization, the number of directory entries is exactly n, the number of keys in the data base. When using Wong and Chiang's scheme, many of the 2^n possible atomic sets are empty; the number of nonempty atomic sets may however still be larger than n in most cases. Furthermore, the directory searches may be very time-consuming for a query. Consider a query $K_1 \wedge K_{10} \wedge K_{50}$, a fully developed disjunctive normal form of this query contains 2^{n-3} products, each of which corresponds to an atomic set --- empty or not. This implies that 2^{n-3} directory look-ups must be performed even though many of them are unsuccessful searches (corresponding to those empty atomic sets). One should note that the value of n may be fairly large in a data base, and 2^{n-3} is much larger.

Neither of the above two traditional look-up procedures seems to be feasible for Wong and Chiang's scheme. Unless a fast look-up method is developed for this scheme, it is unclear how to use this method in practice.

As to file maintenance operations, some are very easy and some are very hard. If, for example, a key is deleted from or added to a data record r, only two atomic sets need to be modified, namely, the one to which record r was originally assigned and the one to which record r is to be transferred. This is because each record appears in exactly one atomic set. However, when a new key is added to the data base, the number of possible atomic sets grows from 2^n to 2^{n+1}. Many atomic sets must be broken up into two, and the directory --- if one is used --- must be globally reorganized. Similarly, when a key is deleted from the data base, certain atomic sets may have to be combined.

In summary, Lum's method makes it possible to retrieve all qualified record in <u>one</u> access, but extra costs are incurred in file maintenance and storage redundancy. Wong and Chiang's scheme avoids list intersections, but the large number of atomic sets in a query cannot be looked up efficiently. Both approaches do not really lead to practically usable combined indices.

In the remainder of this chapter, a heuristic approach to the combined index problem is proposed. It does reduce considerably the number of list accesses required for query processing, although not as much as in Lum's method; it also decreases the number of list intersection operations but cannot completely eliminate them. At the same time fast update operations and relatively modest storage requirement are produced.

Section 4.2 described the structure and processing procedures for an inverted list organization using combined indices. Section 4.3 proposes some methods to generate combined indices for inverted lists and also presents experimental results. Section 4.4 and 4.5 describe the same problems for a multilist organization.

4.2 Structure and Processing Algorithms for Combined Indices in Inverted List Organizations

Instead of constructing combined indices for all possible groups of keys as in Lum's or Wong and Chiang's methods, the proposed heuristic approach selects a group of keys for a combined index only when these keys co-occur frequently in the queries. In most information systems, it is very likely that special patterns of keyword usage will develop. Thus, groups of keys may co-occur repeatedly in queries because these keys may jointly specify some important concept which is common to many users. Hence, the heuristic algorithms assume that usage statistics can be collected for the keys, and combined indices are constructed only for those keys whose co-occurrences patterns in the queries are sufficiently frequent. In this section, the structure and processing algorithms for combined indices are discussed before the index selecting algorithms are introduced in section 4.3. A presentation in this order can better motivate the design rationale of the heuristic index selecting algorithms.

Structure of a combined index

Suppose that k keys K_1, K_2, ..., K_k are selected to form a combined index $C(K_1, K_2, \cdots, K_k)$. Only a single combined list is then stored for the k keys jointly instead of the k separate inverted lists. The structure of a combined list is shown in Figure 4.3a. The combined list consists of three portions. The header portion stores the k keys explicitly. The pointer portion contains $2^k - 1$ pointers P_1, P_2, $\cdots$, P_{2^k-1} to be explained later. The third portion is the identifier portion and it is subdivided into $2^k - 1$ sublists L_1, L_2, $\cdots$, L_{2^k-1}. As illustrated in Figure 4.3b, sublist L_i contains record identifiers $r_{i,1}$, $r_{i,2}$, $\cdots$, r_{i,n_i}, each of which identifies a data record that satisfies the associated qualifying condition on the k keys. Since all the $2^k - 1$ qualifying con-

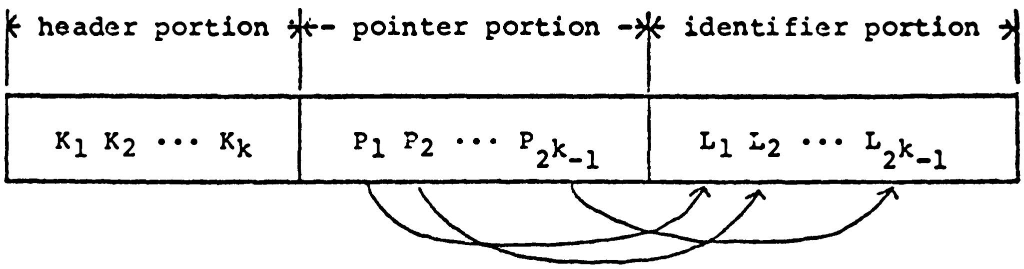


Figure 4.3a A combined inverted list for a combined index $C(k_1, K_2, \cdots, K_k)$

sublist name	qualifying condition of record identifiers in this sublist	record identifiers in this sublist
L_1	$K_1 \wedge K_2 \wedge \cdots\ K_{k-1} \wedge K_k$	$r_{1,1} r_{1,2} \cdots\cdots r_{1,n_1}$
L_2	$K_1 \wedge K_2 \wedge \cdots\ K_{k-1} \wedge \overline{K}_k$	$r_{2,1} r_{2,2} \cdots\cdots r_{2,n_2}$
L_3	$K_1 \wedge K_2 \wedge \cdots\ \overline{K}_{k-1} \wedge K_k$	$\vdots$
$\vdots$	$\vdots$	$\vdots$
L_{2^k-1}	$\overline{K}_1 \wedge \overline{K}_2 \wedge \cdots\ \overline{K}_{k-1} \wedge K_k$	$r_{2^k-1} \cdots\cdots\cdots r_{2^k-1,n_{2^k-1}}$

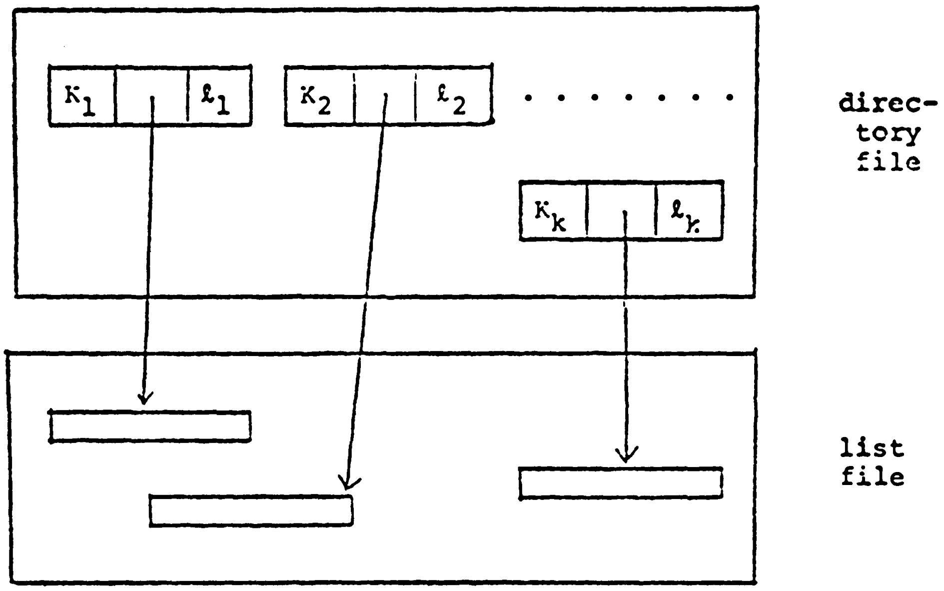
Figure 4.3b Record identifiers contained in each sublist.

ditions are mutually exclusive, the $2^k - 1$ sublists are also pairwise disjoint. It is possible that some sublists are empty; no spaces are then allocated for these sublists. Pointer P_i points to the beginning position of L_i. If L_i is empty, P_i can be set to zero as a special indication.

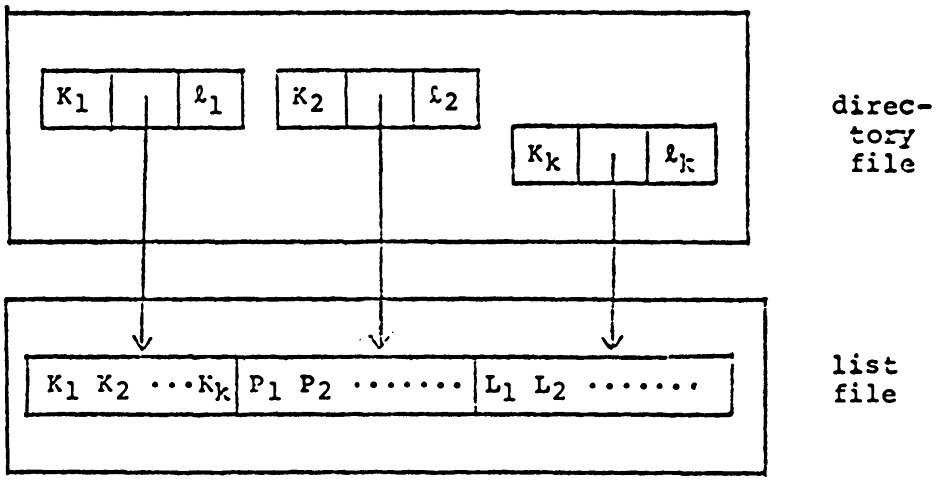
Given a Boolean product composed of a subset of keys from $\{K_i, i = 1, 2, \cdots, k\}$, say $K_1 \wedge K_2 \wedge \cdots \wedge K_j$ where $j < k$, the set of record identifiers satisfying $K_1 \wedge K_2 \wedge \cdots \wedge K_j$ can be easily reconstructed from several sublists. For example, the list for $K_1 \wedge K_2 \wedge \cdots \wedge K_{k-1}$ can be obtained by merging the sublist for $K_1 \wedge K_2 \wedge \cdots \wedge K_{k-1} \wedge K_k$ and the sublist for $K_1 \wedge K_2 \wedge \cdots \wedge K_{k-1} \wedge \overline{K}_k$.

As shown in Figure 4.4, the combined list for $C(K_1, K_2, \cdots, K_k)$ is stored in the list file of an inverted list organization. It is pointed to by all of the k directory entries corresponding to $K_1, K_2, \cdots$, and K_k. So, whenever a query contains one or more keys in $\{K_i, i = 1, 2, \cdots, k\}$, this combined inverted list will be retrieved and the required list for the query can be obtained by merging certain sublists in the combined list.

The picture can be more complicated if a key K_i may participate in more than one combined index. More pointers --- one for each combined index of which K_i is a member --- must be stored in the directory entry of K_i. The list of record identifiers pertaining to K_i will have repeated appearances



k separate inverted lists when no combined index techniques are used in an inverted list organization.



One combined inverted list for the k keys $K_1, K_2, \cdots, K_k$ if $C(K_1, K_2, \cdots, K_k)$ is a combined index. Note that in both figures the data files are not shown.

Figure 4.4

in several combined lists. As a result, redundant storage space is needed and any update operation involving K_i requires duplicated effort. The problems seem to suggest that one should adopt the restriction that any key is allowed to become a part of at most one combined index. Since only frequently co-occuring keys are collected in a combined index, and since several keys may be included in one combined index, at most a marginal effect is produced by assigning a key to more than one combined index. The restriction of assigning each key to at most one combined index is thus adopted for the algorithms proposed in the next section.

The storage requirement of combined indices should also be examined. For a combined index $C(K_1, K_2, \cdots, K_k)$, a single combined list replaces the k individual lists (Figure 4.4). Since the combined list contains the union of the k lists of record identifiers corresponding to keys $K_1, K_2, \cdots$, and K_k, the length of the combined list is smaller than the sum of the lengths of the k individual lists. The header portion and pointer portions of the combined list do not take up too much space so long as k is not very large. Hence, in general, the storage space needed by the combined indices may be fairly small.

<u>Querying Procedure</u>

Assume that a combined index $C(K_1, K_2, \cdots, K_k)$ is created.

Suppose a query $K_1 \wedge K_2 \wedge \cdots \wedge K_k$ uses all k keys in $C(K_1, K_2, \cdots, K_k)$. When any one of the k keys is decoded, the combined list is retrieved- The header portion of the list --- containing k keys --- indicates to the system that the decoding and list accesses for the remaining k - 1 keys can be bypassed. Moreover, the k - 1 list intersection operations that would be needed if $C(K_1, K_2, \cdots, K_k)$ did not exist are also eliminated.

Normally, a query may not consist of a product of exactly the k keys $\{K_1, K_2, \cdots, K_k\}$ included in a combined index. Let a query product be $K_1 \wedge K_2 \wedge \cdots \wedge K_j \wedge K_{k+1} \wedge K_{k+2} \wedge \cdots \cdots \wedge K_m$ where j is less than k. That is, only j of the k keys included in the combined index are used in the query product. Such queries still benefit by using $C(K_1, K_2, \cdots, K_k)$. In this case, the set of record identifiers satisfying $K_1 \wedge K_2 \wedge \ldots \wedge K_j$ can be retrieved in one access - a saving of j - 1 key decodings and the same number of list accesses and intersections.

The individual keys in a combined index may be distributed among several Boolean products in the disjunctive normal form of a query. For example, query $(K_1 \wedge K_2 \wedge \cdots \wedge K_{a-1} \wedge K_a \wedge K_{k+1}) \vee (K_{a+1} \wedge K_{a+2} \wedge \cdots \wedge K_j \wedge K_{k+2})$ also contains $j(<k)$ keys --- $K_1, K_2, \ldots, K_{a-1}, K_a, K_{a+1}, \ldots, K_j$ --- belonging to $C(K_1, K_2, \ldots, K_j, \ldots, K_k)$. If $K_1, K_2, \ldots, K_k$ are

all simple indices, a - 1 and j - a - 1 list intersections would be performed to obtain the record identifier lists for $K_1 \wedge K_2 \ldots \wedge K_a$ and $K_{a+1} \wedge \ldots \wedge K_j$ respectively; also j key decodings and j list accesses are thus necessary. By using the combined index $C(K_1, K_2, \ldots, K_k)$, one can save j - 2 list intersection operations and reduce the number of key decodings and list accesses by j - 1 each. Comparing this with the previous case where the j keys are in the same query product, the same numbers of key decodings and list accesses are saved, but the number of list intersection operations saved is down by one.

To summarize, one can draw the following conclusion : In comparison with the organization where no combined indices are constructed, a query using j of the k keys in a combined index, can always be processed with a reduction of j - 1 key decodings and j - 1 list accesses, no matter how the j keys are related to each other in the query; however, the number of list intersection operations saved may be anywhere between zero and j - 1 depending on the way in which the j keys are used in the query.

File Maintenance Procedures

The file maintenance operations are easy to perform because each key is either a simple index or the member of only one combined index. Consider the following five types of update operations .

1. <u>Add a key K_j to the data record with identifier r.</u>
   If K_j is a simple index, it is only necessary to add r to the simple inverted list corresponding to key K_j. Suppose that K_j is a member of a combined index $C(K_1, K_2, \ldots, K_j, \ldots K_k)$ and that record r contains i keys $\{K_a, a = 1, 2, \ldots i\}$ in the combined index where $0 \leq i \leq k - 1$. Let L_1 and L_2 be two sublists in the combined list for $C(K_1, K_2, \ldots, K_k)$. Also let the qualifying conditions of the record identifiers in L_1 and L_2 be $K_1 \wedge K_2 \wedge \cdots \wedge K_i \wedge \overline{K}_{i+1} \wedge \cdots \wedge \overline{K}_k$ and $K_1 \wedge K_2 \wedge \cdots \wedge K_i \wedge \overline{K}_{i+1} \wedge \overline{K}_{i+2} \wedge \cdots \wedge K_j \wedge \overline{K}_{j+1} \wedge \cdots \wedge \overline{K}_k$ respectively. Before key K_j is added to record r, the identifier should be associated with L_1. To add key K_j to record r, it is only necessary to remove the identifier r from the sublist L_1 and insert it into the sublist L_2. This procedure involves one reading of the combined list and one rewriting of the combined list after modification --- the same as when K_j is a simple key.
2. <u>Delete a key K_j from the data record with identifier r.</u>
   Again, it is only necessary to consider the case when K_j is in a combined index $C(K_1, K_2, \cdots, K_j, \cdots, K_k)$. Let L_1 and L_2 be the same sublists as defined previously. If record r originally contains keys $K_1, K_2, \cdots, K_i$ and K_j in the combined index $C(K_1, K_2, \cdots, K_k)$, then deleting K_j from r only necessitates the transfer of identifier r from

sublist L_2 to L_1.

3. <u>Modify a non-key data field of a data record.</u>
   This operation does not involve any index keys.
4. <u>Add a record of y keys to the data base.</u>
   Let the record have identifier r and the y keys be K_1, K_2, $\cdots$, K_y. If any of the y keys is a simple index, the modification algorithm is the same as the one given in section 3.2.1. Assuming that K_1, K_2, $\cdots$, K_i of record r are contained in a combined index $C(K_1, K_2, \cdot\cdot, K_i, K_{y+1}, K_{y+2}, \cdots, K_k)$, then identifier r should be added to the sublist qualified by $K_1 \wedge K_2 \wedge \cdots \wedge K_i \wedge \bar{K}_{y+1} \wedge \bar{K}_{y+2} \wedge \cdots \wedge \bar{K}_k$. This requires only one read and one rewrite of the combined list. If the combined index $C(K_1, .., K_i, K_{y+1}, \cdots, K_k)$ does not exist, all K_1, K_2, .., K_i must be simple lists, and i list reads and i list rewrites would be required. So, using combined indices, this type of update operation may actually be quite fast.
5. <u>Delete a record of y keys from the data base.</u>
   This procedure is essentially parallel to that of a type 4 operation. The possible reduction in cost is the same as before.

In summary, the update cost of an inverted list organization using combined indices is lower than the corresponding cost for an inverted list organization without combined indices.

4.3 Algorithms and Experiments for Combined Indices in Inverted List Organizations

The first subsection proposes two heuristic algorithms for combined indices. Both methods are based on the assumption that a set of frequently used queries is collected and that the frequencies of these queries are known. Combined indices are created to reduce the retrieval cost for this set of queries. Section 4.3.2 shows experimental results for both algorithms.

4.3.1 Heuristic Algorithms

Let $Q_1, Q_2, \cdots, Q_m$ be a set of frequently used queries in a data base. The frequency of query Q_i is denoted by f_i for $i = 1, 2, \cdots, m$. The keys used in each query are indicated by an access matrix defined as follows :

Definition An access matrix Z is an m x n binary matrix such that $Z_{i,j} = 1$ if and only if query Q_i uses key K_j. The j^{th} column of Z, denoted by Z(j), is called the access vector of key K_j. Z(j) specifies the queries which include K_j in the retrieval expressions.

When a combined index $C(K_1, K_2, \cdots, K_k)$ is created, any query whose retrieval expression contains any of $\{K_1, K_2, \cdots, K_k\}$ should access $C(K_1, K_2, \cdots, K_k)$. It is convenient to represent this fact by also defining an access vector for each combined index.

Definition The access vector $Z(1, 2, \cdots, k)$ for a combined index $C(K_1, K_2, \cdot\cdot, K_k)$ is the logical OR of the access vectors of the k component keys. That is, $Z(1, 2, \ldots, k) = Z(1) \vee Z(2) \vee \cdots \vee Z(k)$.

It is the purpose of this discussion to formulate the cost equations which apply to the processing of combined indices by using the above definitions. As pointed out in section 4.2, the combined indices reduce the querying costs by decreasing the number of key decodings, list accesses, and list intersections. In the following calculations, the cost of the list intersections is ignored for simplicity because these are carried out in main memory and are thus much faster than the other two types of actions.

Following the convention of Chapter 3, let C_{decode} be the average cost required to decode a key and let the cost to retrieve an inverted list of length L be $T_L + T_t L$, where T_L is the aver[illegible] time and T_t is the time needed to transmit one byte of data from peripheral devices to main memory. The following example illustrates the decrement in querying costs due to the combined indices :

<u>Example</u> : Let $C(K_1, K_2)$ be a combined index consisting of two keys. Let L_1, L_2 and L_{12} be the lengths of the inverted lists for key K_1, key K_2 and index $C(K_1, K_2)$. Consider a simplified case where there are only four queries Q_1, Q_2, Q_3, Q_4 with frequencies f_1, f_2, f_3 and f_4. Let the access matrix Z be

$$\begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix}.$$

So, query Q_1 uses neither of K_1 and K_2 and query Q_2 uses K_2 but not K_1. etc.. The existence of the combined index $C(K_1, K_2)$ does not affect the querying efficiency of Q_1 because Q_1 uses neither of the two keys. For Q_2, the cost to retrieve the K_2 list would be $C_1 = [C_{decode} + (T_L + T_t L_2)]\ f_2$, when K_2 is a simple index. When K_2 is a component of $C(k_1, K_2)$, the combined list for $C(k_1, K_2)$ must be retrieved for Q_2; from this the K_2 list is easily obtained. The cost to do so is $C_2 = [C_{decode} + (T_L + T_t\ L_{12})]\ f_2$. Comparing C_2 with C_1, cost C_2 is slightly higher because L_{12} is larger than L_2. Thus, an increment in querying cost results for Q_2. Fortunately, this increment is very small since the dominant terms in C_1 and C_2 are C_{decode} and T_L. The querying cost for Q_3 also increases slightly for the same reason. The actual gain by creating $C(k_1, K_2)$ lies with query Q_4 which uses both keys. Without the combined index, the cost for Q_4 is $f_4\ [2\ C_{decode} + (T_L + T_t\ L_1) + (T_L + T_t\ L_2)]$ while it reduces to $f_4\ [C_{decode} + (T_L + T_t\ L_{12})]$ by using $C(K_1, K_2)$.

In this example, K_1 and K_2 are assumed to be simple indices. If K_1 and/or K_2 are combined indices themselves, the same conclusions still apply. The following definition states these conclusions in more formal terms :

<u>Definition</u> Let $Z(1)$ and $Z(2)$ be the access vectors for indices

$C_1(K_{i,1}, K_{i,2}, \ldots, K_{i,a})$ and $C_2(K_{j,1}, K_{j,2}, \ldots, K_{j,b})$. The decrement in querying cost obtained by combining these two indices is

$$\begin{aligned} d(C_1, C_2) = \sum_{i=1}^{m} & Z_i(1)\ Z_i(2)\ f_i\ [C_{decode} + T_L + T_t\ (L_1 \\ & + L_2 - L_{12})] \\ & - \sum_{i=1}^{m} (1 - Z_i(1))\ Z_i(2)\ f_i\ [T_t\ (L_{12} - L_2)] \\ & - \sum_{i=1}^{m} Z_i(1)\ (1 - Z_i(2))\ f_i\ [T_t\ (L_{12} - L_1)] \end{aligned}$$

where $Z_i(1)$ is the i^{th} component of vector $Z(1)$.

Note that the first term in $d(C_1, C_2)$ represents the decrement in cost for queries of type Q_4 in the previous example. The second and third terms represents the increments in costs for queries of types Q_2 and Q_3 in the example. There is a net gain by combining two indices into one only if $d(C_1, C_2)$ is positive.

It is clear from equation $d(C_1, C_2)$ that a combined index of k keys can be most effective if every query either uses all of the k component keys or uses none of the k keys. This however is not likely to be the case in practice. The problem then is how to select a collection of combined indices such that the total cost for the m queries as a whole is minimized. An optimal solution to this problem has not been obtained so far, and is likely to be a very hard combinatorial problem.

While the optimal solutions may be difficult, heuristic approaches often produce near-optimal results. Hence, a few heuristic algorithms for combined indices are suggested. Experimental results are collected to show their usefulness.

One constraint imposed on all the heuristic algorithms to be discussed is that the number of keys, k, in a combined index should not be excessively large. This is because the pointer portion of the combined list (Figure 4.3) has a length proportional to $2^k - 1$. When k is large, the overhead due to these pointers becomes unreasonable. Hence, an upperbound MAXKEY is adopted to limit the number of keys in any combined index.

An intuitive algorithm consists in iteratively combining two indices (simple or combined), C_i and C_j, in such a way that the maximum possible decrement $d(C_i, C_j)$ is achieved. This is however a very expensive procedure. For a set of n keys, n^2 operations are needed in each iteration; hence the total number of operations required in the process is of order n^3. For most information systems, the number of keys, n, is usually large. An n^3 algorithm may be prohibitively expensive.

A second approach is to construct one combined index at a time. Each combined index is initialized by first assigning an arbitrary key to it. Then, up to MAXKEY-1 keys are selected for this combined index in successive iterations.

Each selection of these keys is the one which causes maximum decrement in querying cost. The creation of a combined index is completed either when MAXKEY keys are assigned or when no key can be added with a positive cost decrement. The same process is then repeated for the creation of the next combined index. More formally, this algorithm can be stated as follows :

Algorithm I Given a set of queries $Q = \{Q_1, Q_2, \ldots, Q_m\}$ and a set of keys $K = \{K_1, K_2, \ldots, K_n\}$. Let f_i be the frequency of Q_i and let C denote the combined index under construction. Let Z be the access matrix and MAXKEY be the maximum number of keys allowable in any combined index. The output is a set of combined indices.

Step 1 : If K is empty then STOP.

(Steps 2 through 6 create a combined index C)

Step 2 : (Initialize the combined index C) Assign to C an arbitrary key K_a of set K. $K \leftarrow K - \{K_a\}$.

Step 3 : Repeat steps 4 and 5 for i = 1, 2,···, MAXKEY-1.

(At each iteration, assign to C the key which produces maximum decrement in querying cost)

Step 4 : Select a key K_{bi} from K such that $d(C, K_{bi}) = \max\{d(C, K_j) \mid K_j \in K\}$.

Step 5 : If $d(C, K_{bi}) \leq 0$ then go to step 6; otherwise assign K_{bi} to C, $K \leftarrow K - \{K_{bi}\}$.

Step 6 : Output combined index C.

Step 7 : $C \leftarrow$ empty set. Go to step 1.

Except for step 4, every other step takes constant time. At step 4, each member remaining in set K is to be examined hence it is an O(n) operation. Since step 4 will be executed for a total of n times, the running time of Algorithm I is $O(n^2)$.

Algorithm I may be speeded up if less work is done at step 4. Step 4 chooses a key from all the remaining keys in set K such that the selected key is guaranteed to produce maximum decrement in querying cost. But this is probably not necessary for most practical purposes. If one selects a key which produces a near maximum decrement in cost at less efforts, one may be able to produce combined indices faster without sacrificing system performance. Consider two groups of keys in the system, say group A and group B. If the keys in group A co-occur frequently with other keys in the same group, but co-occur seldom with keys in group B, then it is not likely for any combined index constructed by algorithm I to contain keys from both groups. In other words, when a combined index under construction contains keys from group A, one does not really need to check the keys in group B.

The second heuristic algorithm for combined indices, Algorithm II, makes use of such an idea. It consists of two phases. For phase one, a classification algorithm is applied to the set of keys in the system in order to produce groups of keys with the desirable property that only keys within the same group are likely to be members of a combined index. Phase two applies algorithm I to each of the groups to actually create the combined indices.

The running time of such a procedure can be estimated easily. Suppose that there are n keys in the system and that the classification algorithm produces groups of limited size between S_1 and S_2, where S_1 and S_2 are constants and $S_1 < S_2$. The number of groups produced is at most n / S_1 which is $O(n)$. The time needed to run algorithm I for each group is at most $O(S_2^{\,2})$. Assuming that $f(n)$ is the classification time, the total running time of Algorithm II is therefore equal to $f(n) + (n / S_1) O(S_2^{\,2})$ which can in turn be written as $f(n) + O(n)$ since S_1 and S_2 are constants. Expression $f(n)$ depends on the clustering algorithm used. As pointed out in section 2.6.6, Williamson (1974) proposed a GROTRE algorithm which has a worst as well as average case complexity $O(n \log n)$. If the GROTRE algorithm is used in Algorithm II, its running time is then equal to $O(n \log n) + O(n)$ which is $O(n \log n)$. The $O(n \log n)$ time bound is much better than the $O(n^2)$ time bound of Algorithm I.

Algorithm II can be summarized as follows :

<u>Algorithm II</u>

<u>Phase one</u> : Use the GROTRE clustering algorithm to produce groups of keys $G_1, G_2, \cdots, G_g$ where the number of keys in each group is between S_1 and S_2.

Phase two : Apply algorithm I on each group separately to create combined indices.

4.3.2 Experimental Results

The CRANFIELD query collection used with the SMART retrieval system contains 155 queries. A subset of 40 queries is used for the experiments. There are on average nine keys in a query. The total number of distinct keys in the forty queries is 177. This query set constitutes a 40 by 177 access matrix Z to be used for the algorithms. Each query is assigned a frequency which is uniformly distributed in the range from 5 to 10.

The querying costs are measured in terms of the amount of I/O time required for key decodings and inverted lists retrieval. Following the assumptions of chapter three, the directory is a B-tree structure and is stored in an IBM 2305 drum which has an average latency time of 2.5 msec. The lists are stored in an IBM 2314 disk of average latency time 12.5 msec.

Algorithm I is applied to the query set with MAXKEY, the maximum number of keys allowable in any combined index, equal to 5. The result is shown in Table 4.1. When combined indices are used, the I/O time is almost reduced by half.

Algorithm II is tested on the same set of data. Among the 177 distinct keys in the forty queries, 99 keys appear

	with combined indices	without combined indices	without / with
number of key decodings or number of list accesses	1321	2925	2.21
list transmission time in msec	4932	4425	0.90
total I/O time in msec	24747	48300	1.95

Table 4.1 Performance of combined indices created by Algorithm I. MAXKEY = 5.

	combined indices created by Algorithm I	combined indices created by Algorithm II	Algorithm I / Algorithm II
number of key decodings or number of list accesses	1321	1336	0.988
list transmission time in msec	4932	4960	0.994
total I/O time in msec	24747	25000	0.990

Table 4.2 Performance comparison of the combined indices created by Algorithm I and Algorithm II. MAXKEY = 5.

in one query each. Hence Williamson's clustering algorithm is applied to the remaining 78 keys only. This produces two clusters. Each of the 99 keys is then assigned to a more closely related cluster of the two.

The performance of the combined indices created by Algorithms I and II are compared in Table 4.2. The combined indices created by Algorithm II are less effective than those created by Algorithm I by only one percent. Since Algorithm II is much faster than Algorithm I, the remainder of the experiments are conducted by using Algorithm II.

The combined indices are created using a number of specific queries and query frequencies collected during a period of time. In a real system, the queries and frequencies do not remain unchanged forever. If after a period of time, the queries and/or frequencies become different from those used in the creation of combined indices, the effectiveness of these combined indices should be investigated. For this purpose, four different uniform distributions --- F_1, F_2, F_3 and F_4 --- are used to test the performance of the combined indices. F_1 is the frequency distribution used in the creation of combined indices. F_2 and F_4 have frequency ranges (1,1) and (1,10) respectively. F_3 has the same range (5,10) as F_1. The frequency assignments for each of the forty queries are however different in F_1 and F_3. Thus, F_1 represents the test case where the query frequencies in the system remain unchanged from those used in the creation of the combined indices.

F_2, F_3 and F_4 are used to test the effectiveness of the combined indices when the query frequencies alter. Table 4.3 compares the performances of combined indices for the four test cases when MAXKEY is equal to 2. If the query frequencies remain unchanged, i.e. case F_1, the I/O time needed when no combined indices are used is greater by a factor of 1.465 compared with the I/O time where there are combined indices. In the case of F_2, F_3 and F_4 where the query frequencies are different, the ratio of I/O time deteriorates only slightly to 1.462, 1.461 and 1.449 respectively. Therefore, the performance of the combined indices is not perturbed seriously due to minor changes in query frequencies. Table 4.4 through 4.8 show similar results when the value of parameter MAXKEY increases from 3 to 7. When more and more keys are allowed in a combined index, less and less key decodings and list accesses are required. But the value of MAXKEY cannot grow indefinitely because the overhead due to the key and pointer portions of the combined list can become prohibitively large.

4.4 Structure and processing Algorithms for Combined Indices in Multilist Organizations

Whether combined indices are used or not, an inverted list process retrieves only those data records which satisfy the query criterion exactly. Hence, the combined indices used with inverted list organizations simply reduce the number of

Measurements \ Query frequency		F_1	F_2	F_3	F_4
Number of key decodings or Number of list accesses	with combined indices	1909	240	1878	1577
	without combined indices	2925	367	2869	2386
	$\frac{\text{without}}{\text{with}}$	1.532	1.529	1.528	1.513
lists transmission time in msec	with combined indices	4332	549	4333	3502
	without combined indices	4425	561	4435	3555
	$\frac{\text{without}}{\text{with}}$	1.022	1.022	1.024	1.015
total I/O time in msec	with combined indices	32967	4149	32403	27157
	without combined indices	48300	6066	47470	39345
	$\frac{\text{without}}{\text{with}}$	1.465	1.462	1.461	1.449

<u>Table 4.3</u> Performance for the combined indices created by Algorithm II. MAXKEY = 2.

Measurements \ Query frequency		F_1	F_2	F_3	F_4
Number of key decodings or Number of list accesses	with combined indices	1608	203	1589	1355
	without combined indices	2925	367	2869	2386
	$\frac{\text{without}}{\text{with}}$	1.819	1.808	1.806	1.761
lists transmission time in msec	with combined indices	4597	582	4593	3724
	without combined indices	4425	561	4435	3555
	$\frac{\text{without}}{\text{with}}$	0.963	0.964	0.966	0.955
total I/O time in msec	with combined indices	28717	3627	28428	23749
	without combined indices	48300	6066	47470	39345
	$\frac{\text{without}}{\text{with}}$	1.682	1.673	1.670	1.657

Table 4.4 Performance of combined indices created by Algorithm II. MAXKEY = 3.

Measurements \ Query frequency		F_1	F_2	F_3	F_4
Number of key decodings or Number of list accesses	with combined indices	1400	177	1381	1150
	without combined indices	2925	367	2869	2386
	$\frac{\text{without}}{\text{with}}$	2.089	2.073	2.078	2.075
lists transmission time in msec	with combined indices	4522	574	4507	3664
	without combined indices	4525	561	4435	3555
	$\frac{\text{without}}{\text{with}}$	1.001	0.977	0.984	0.970
total I/O time in msec	with combined indices	25521	3229	25221	20914
	without combined indices	48300	6066	47470	39345
	$\frac{\text{without}}{\text{with}}$	1.893	1.879	1.882	1.881

Table 4.5 Performance of combined indices created by Algorithm II. MAXKEY = 4.

Measurements \ Query frequency		F_1	F_2	F_3	F_4
Number of key decodings or Number of list accesses	with combined indices	1336	167	1294	1079
	without combined indices	2925	367*	2869	2386
	$\frac{\text{without}}{\text{with}}$	2.189	2.198	2.217	2.211
lists transmission time in msec	with combined indices	4960	625	4873	3941
	without combined indices	4425	561	4435	3555
	$\frac{\text{without}}{\text{with}}$	0.892	0.898	0.910	0.902
total I/O time in msec	with combined indices	25000	3130	24283	20126
	without combined indices	48300	6066	47470	39345
	$\frac{\text{without}}{\text{with}}$	1.932	1.938	1.955	1.946

Table 4.6 Performance of combined indices created by Algorithm II. MAXKEY = 5.

Measurements \ Query frequency		F_1	F_2	F_3	F_4
Number of key decodings or Number of list accesses	with combined indices	1259	159	1237	1038
	without combined indices	2925	367	2869	2386
	without / with	2.323	2.308	2.319	2.299
lists transmission time in msec	with combined indices	5283	666	5224	4248
	without combined indices	4425	561	4435	3555
	without / with	0.838	0.842	0.849	0.836
total I/O time in msec	with combined indices	24168	3051	23779	19817
	without combined indices	48300	6066	47470	39345
	without / with	1.998	1.988	1.996	1.985

Table 4.7 Performance of combined indices created by Algorithm II. MAXKEY = 6.

Measurements \ Query frequency		F_1	F_2	F_3	F_4
Number of key decodings or Number of list accesses	with combined indices	1149	145	1127	960
	without combined indices	2925	367	2869	2386
	$\frac{\text{without}}{\text{with}}$	2.546	2.531	2.546	2.485
lists transmission time in msec	with combined indices	5651	713	5555	4631
	without combined indices	4425	561	4435	3555
	$\frac{\text{without}}{\text{with}}$	0.783	0.787	0.798	0.768
total I/O time in msec	with combined indices	22886	2887	22460	19031
	without combined indices	48309	6066	47470	39345
	$\frac{\text{without}}{\text{with}}$	2.110	2.101	2.114	2.067

Table 4.8 Performance of combined indices created by Algorithm II. MAXKEY = 7.

key decodings and list accesses. The situation is different for multilist organizations. A separate list file does not exist in a multilist organization and the data records retrieved may not exactly satisfy the query criterion. So, the structure of the combined indices may reduce the number of key decodings as well as the number of data records that are actually retrieved.

<u>Structure of a combined index</u>

Assume that k keys $K_1, K_2, \ldots, K_k$ are chosen to form a combined index $C(K_1, K_2, \ldots, K_k)$. An <u>address bucket</u> is constructed for this combined index as shown in Figure 4.5a. The bucket consists of a <u>header portion</u> and a <u>pointer portion</u>. The header portion contains the k member keys of the index. Each of the $2^k - 1$ pointers points to the head of a certain multilist. Figure 4.5b illustrates the qualifying condition for the data records in each of the $2^k - 1$ multilists. A length field L_i is attached to each pointer P_i to indicate the number of data records in the corresponding multilist. Figure 4.6 depicts the address bucket for a combined index of two keys $C(K_1, K_2)$. The address buckets are stored at the end of the directory and are pointed to by the key entries of K_1 and K_2. P_1 in the address bucket points to the first data record containing both K_1 and K_2. Pointers P_2 and P_3 point to the header records of the multilists corresponding to the qualifying conditions $K_1 \wedge \overline{K}_2$ and $\overline{K}_1 \wedge K_2$ respectively.

← header portion →	← pointer portion →
$K_1\ K_2 \cdots\cdots K_k$	$P_1\ L_1\ P_2\ L_2 \cdots P_k\ L_k$

Figure 4.5a An address bucket for a combined index $C(K_1, K_2, \cdots, K_k)$ in multilist organization.

pointer	qualifying condition of data records contained in the mulsilist pointed to by P_i
P_1	$K_1 \wedge K_2 \wedge \cdots \wedge K_{k-1} \wedge K_k$
P_2	$K_1 \wedge K_2 \wedge \cdots \wedge K_{k-1} \wedge \overline{K}_k$
P_3	$K_1 \wedge K_2 \wedge \cdots \wedge \overline{K}_{k-1} \wedge K_k$
.	
.	
.	
.	
.	
P_{2^k-1}	$\overline{K}_1 \wedge \overline{K}_2 \wedge \cdots \wedge \overline{K}_{k-1} \wedge K_k$

Figure 4.5b Qualifying conditions for data records in the $2^k - 1$ possible multilists in a combined index $C(K_1, K_2, \cdots, K_k)$.

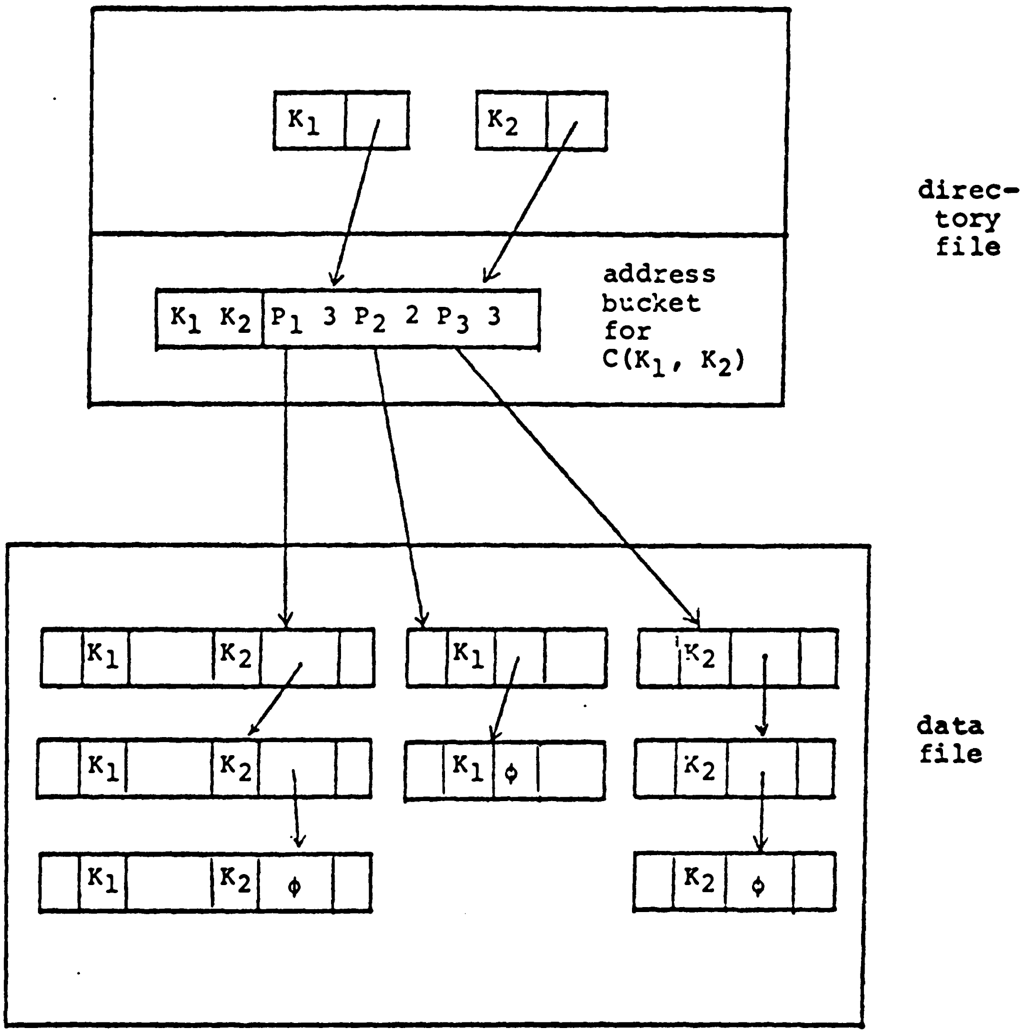


Figure 4.6 A multilist organization with combined indices.

Querying procedure

The querying procedure using combined indices is straightforward. Assume that a combined index $C(K_1, K_2, \cdots, K_k)$ is created. Consider a query $K_1 \wedge K_2 \wedge \cdots \wedge K_j \wedge K_{k+1} \wedge K_{k+2} \wedge \cdots \wedge K_m$ which contains j of the k keys in $C(K_1, K_2, \cdots, K_k)$. Assume that $K_{k+1}, K_{k+2}, \cdots, K_m$ are all implemented as simple indices. When the anyone of the j keys $K_1, K_2, \cdots, K_j$ is first decoded, the address bucket of $C(K_1, K_2, \cdots, K_k)$ is retrieved. The header portion of the address bucket dictates that decodings for the other j - 1 keys be bypassed. This saves j - 1 key decodings for the query but one additional access to the address bucket is needed. The length of the multilist corresponding to the qualifying condition $K_1 \wedge K_2 \wedge \cdots \wedge K_j$ is then available from the address bucket, and this length is compared with the lengths of the multilists for $K_{k+1}, K_{k+2}, \cdots$, and K_m. The data records in the shortest list are retrieved to answer the query.

Since the list corresponding to $K_1 \wedge K_2 \wedge \cdots \wedge K_j$ is the intersection of j multilists, it is likely that this list is very short. Consequently, few data records need to be retrieved. On the other hand, let a query contain only one of the k keys in $C(K_1, K_2, \cdots, K_k)$. The existence of this combined index can then make the retrieval cost for this query more expensive than when $C(K_1, K_2, \cdots, K_k)$ is not created. Because of the additional access to the address bucket. Therefore, the combined indices in multilist organizations reduce the number of key decodings as well as the number of data record retrievals, while at the same time extra accesses to the address buckets are demanded.

The accesses to the address buckets need not jepardize the usefulness of combined indices for multilist organizations. In general, the work saved in retrieving fewer data records and in decoding fewer keys will be more than enough to offset the additional work needed in retrieving the address buckets. The experimental results in the next section justify this conjecture.

Extra storage space is indeed necessary for the address buckets. But as long as the number of combined indices is small --- which is true if combined indices are created only for the small subset of keys that co-occur frequently in queries, the storage allocation for the buckets should not cause any problem.

<u>File maintenance procedures</u>

Five types of file maintenance operations are discussed briefly as follows :

1. <u>Add a key K_j to the data record with identifier r.</u>

   Assume that key K_j is a member of the combined index $C(K_1, K_2, \ldots, K_k)$ and record r originally contains i keys $(K_1,$

$K_2, \ldots, K_i$) which are members of $C(K_1, K_2, \ldots, K_k)$. The addition of K_j to record r should move record r from the list qualified by the condition $K_1 \wedge K_2 \wedge \ldots \wedge K_i \wedge \overline{K}_{i+1} \wedge \ldots \wedge \overline{K}_k$ to the list qualified by the condition $K_1 \wedge K_2 \wedge \ldots \wedge K_i \wedge \overline{K}_{i+1} \wedge \ldots \wedge \overline{K}_{j-1} \wedge K_j \wedge \overline{K}_{j+1} \wedge \ldots \wedge \overline{K}_k$. This is equivalent to a key deletion operation plus a key insertion operation and is therefore more complicated than when K_j is not a member of any combined index.

2. <u>Delete a key K_j from the data record with identifier r.</u>
   This is similar to the procedure in a type 1 operation.
3. <u>Modify a non-key data field of a data record.</u>
   This operation does not involve any index keys.
4. <u>Add a record of y keys to the data base.</u>
   If x of the y keys in the record are members of the same combined index, then x - 1 key decodings are saved compared with the case where all these x keys are simple indices.
5. <u>Delete a record of y keys from the data base.</u>
   To delete a data record, it is only necessary to set the record flag as in the case where no combined indices are used.

In summary, compared with a multilist organization with simple indices only, the existence of the combined indices requires more work in performing a type one or a type two update operation but renders type four operations easier. Type three and type five update operations are not affected by the creation of the combined indices.

4.5 Algorithms and Experiments for Combined Indices in Multilist Organizations

Like the case of the inverted list organization, combined indices in multilist organizations will be constructed only for those keys which co-occur in queries. A set of frequently used queries is collected for this purpose. Section 4.5.1 discusses the querying costs incurred with the combined indices and section 4.5.2 presents some experimental results for combined indices in multilist organizations.

4.5.1 Heuristic Algorithm

The access matrix Z and the access vectors for single keys or combined indices are defined as in section 4.3.1. An additional vector N is introduced as follows :

Definition For a set of m queries, N is a vector of m elements such that the i^{th} component of N, denoted by N_i, represents the number of data records that the i^{th} query must retrieve.

If the i^{th} query is a product of k keys, say $K_1 \wedge K_2 \wedge \ldots \wedge K_k$, then N_i is initially equal to the number of records in the shortest of the k multilists corresponding to the k keys. At a later time, when combined indices involving some of these k keys are created, the value of N_i may be updated.

The changes in querying costs due to the creation of combined indices is illustrated by an example.

Example Let $C(K_1, K_2)$ be a combined index consisting of two keys. Consider a system of four queries Q_1, Q_2, Q_3 and Q_4. Suppose that the access matrix A is of the form $\begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix}$.

The creation of the combined index $C(K_1, K_2)$ does not change the querying cost of Q_1 because query Q_1 does not include either K_1 or K_2. The querying costs for Q_2, Q_3 and Q_4 are however affected by the adoption of $C(K_1, K_2)$. Assuming that $C(K_1, K_2)$ does not exist, key K_2 in query Q_2 would be decoded and the directory entry of K_2 would point directly to the head record of the multilist for K_2. When $C(K_1, K_2)$ does exist, the directory entry for K_2 points to the address bucket for $C(K_1, K_2)$ instead (refer to Figure 4.6 in section 4.4), and pointers in the address bucket will in turn point to records containing key K_2. Therefore, the existence of $C(K_1, K_2)$ forces Q_2 to make one additional access to the address bucket. Similarly, an increment in cost by one access to the address bucket also arises for Q_3. Query Q_4 is the only one that may benefit from the construction of $C(K_1, K_2)$. Assume that Q_4 is a product of m keys, $K_1 \wedge K_2 \quad \ldots \wedge K_m$. Let $L(K_i)$ be the number of data records containing key K_i. Then the original value of N_4, the number of data records to be retrieved by Q_4, is equal to $\min\{L(K_i) \mid i = 1, 2, \ldots, m\}$. After the creation of $C(K_1, K_2)$, a separate multilist is formed to connect records containing both K_1 and K_2. Therefore, if $L(K_1 \wedge K_2)$ is smaller than the original value of N_4, the value of N_4 should be replaced by $L(K_1 \wedge K_2)$ --- which implies that fewer data records need to be retrieved for query Q_4. Moreover, one key decoding can be saved for query Q_4 by coalescing K_1 and K_2 into a combined index.

The decrement in querying cost $d(K_1, K_2)$ by combining two single keys K_1 and K_2 can be calculated by following the argument in the example. In case two combined indices C_1 and C_2 are merged to form a single combined index, or when a single key K_1 is assigned to an existing combined index C_1, the decrement in costs, $d(C_1, C_2)$ or $d(C_1, K_1)$, can be calculated in a similar fashion. Expressions for $d(K_1, K_2)$ or $d(C_1, C_2)$ or $d(C_1, K_1)$ are too cumbersome to be written out explicitly. The corresponding cost evaluations are programmed into the heuristic algorithms which produce combined indices for multilist organizations.

The heuristic algorithms in section 4.3.1 for selecting combined indices in inverted list organizations can also be utilized to create the combined indices for multilist organizations. The only alteration to the algorithms is the different calculation of the cost decrement equation $d(C_1, C_2)$ when C_1 and C_2 are combined. The experimental results in section 4.3.2 showed that the combined indices produced by Algorithm I do not really outperform those selected by Algorithm II in an inverted list organization. Therefore, only Algorithm II is used in the next section for the creation of combined indices in multilist organizations.

4.5.2 Experimental Results

The frequency distribution F_1 and the forty queries used in the previous experiments are again used for the creation of combined indices for multilist organizations. Williamson's clustering algorithm classifies the 177 distinct keys in the forty queries into two clusters. Algorithm I described in section 4.3.1 is then applied to keys in each of the clusters to actually generate the combined indices. The querying costs are measured in terms of the number of key decodings, and data record retrievals, and address bucket accesses. The sum of the three measurements is expressed as the total I/O time needed to process these queries. Three more query frequency distributions (F_2, F_3, and F_4) --- which are different from the distribution F_1 used in the creation of the combined indices --- are used to test the validity of the combined indices after minor perturbations have changed query characteristics in the system. The definitions of F_1, F_2, F_3, and F_4 are the same as those in section 4.3.2.

Table 4.9 shows the performance of the combined indices when the maximum number of keys in each index, MAXKEY, is equal to two. When the combined indices are not used, the total I/O time needed to process these queries can be 37 percent more than the I/O time needed when combined indices are used. At various frequency distributions, the percentage improvements are fairly stable. Tables 4.10 to 4.14 represent the performances for combined indices when MAXKEY ranges from 3 to 7. When more keys are included in one combined index, the querying process can be speeded up by a factor of three or four. However, the values of MAXKEY have to be small; otherwise the sizes of address buckets may become too large because the pointer portion of an address bucket grows exponentially in MAXKEY. For the values of MAXKEY that are tested in the experiments, considerable improvements in query processing are achieved by the combined indices.

4.6 Summary of the Combined Index Problem

Some previous approaches to the combined index problem facilitate very fast query retrievals for list structure organizations. However, these methods cause complicated file maintenance operations and require a good deal of redundant storage space. In this chapter, combined indices are restricted to keys that co-occur frequently in queries. In an inverted list organization, the use of such combined indices reduces the number of key decodings and list accesses required for query retrieval. Update operations are also speeded up. Some extra storage space is needed for the headers of the combined indices; but, on the other hand, less space is used for the

Measurements \ Query frequency		F_1	F_2	F_3	F_4
Number of key decodings	with combined indices	2075	260	2035	1717
	without combined indices	2925	367	2869	2386
	$\frac{\text{without}}{\text{with}}$	1.410	1.412	1.410	1.390
Number of data records accessed	with combined indices	2514	303	2325	1900
	without combined indices	3561	438	3417	2820
	$\frac{\text{without}}{\text{with}}$	1.417	1.446	1.470	1.484
Number of address buckets accessed	with combined indices	1935	243	1903	1608
	without combined indices	---	---	---	---
Total I/O time in msec	without combined indices	43034	5241	41093	33297
	with combined indices	58977	6677	52189	43048
	$\frac{\text{without}}{\text{with}}$	1.370	1.270	1.270	1.290

Table 4.9 Performance of combined indices in a multi-list organization.
MAXKEY = 2.

Measurements \ Query frequency		F_1	F_2	F_3	F_4
Number of key decodings	with combined indices	1735	218	1709	1433
	without combined indices	2925	367	2869	2386
	without / with	1.686	1.684	1.679	1.665
Number of data records accessed	with combined indices	1456	169	1247	964
	without combined indices	3561	438	3417	2820
	without / with	2.446	2.592	2.740	2.925
Number of address buckets accessed	with combined indices	1601	201	1572	1327
	without combined indices	---	---	---	---
Total I/O time in msec	with combined indices	27486	3269	24600	19576
	without combined indices	54139	6677	52106	43043
	without / with	1.970	2.040	2.120	2.200

Table 4.10 Performance of combined indices in a multilist organization. MAXKEY = 3.

Measurements \ Query frequency		F_1	F_2	F_3	F_4
Number of key decodings	with combined indices	1589	199	1554	1311
	without combined indices	2925	367	2869	2386
	$\frac{\text{without}}{\text{with}}$	1.841	1.844	1.846	1.820
Number of data record accessed	with combined indices	1137	129	929	697
	without combined indices	3651	438	3417	2820
	$\frac{\text{without}}{\text{with}}$	3.211	3.395	3.678	4.046
Number of address buckets accessed	with combined indices	1523	191	1497	1259
	without combined indices	---	---	---	---
Total I/O time in msec	with combined indices	22731	2671	19844	15590
	without combined indices	55323	6677	52106	43048
	$\frac{\text{without}}{\text{with}}$	2.430	2.500	2.630	2.760

Table 4.11 Performance of combined indices in a multilist organization. MAXKEY = 4.

Measurements \ Query frequency		F_1	F_2	F_3	F_4
Number of key decodings	with combined indices	1413	177	1385	1162
	without combined indices	2925	367	2869	2386
	$\frac{\text{without}}{\text{with}}$	2.070	2.073	2.072	2.019
Number of data record accessed	with combined indices	1042	115	798	586
	without combined indices	3561	438	3417	2820
	$\frac{\text{without}}{\text{with}}$	3.418	3.809	4.282	4.812
Number of address buckets accessed	with combined indices	1331	173	1357	1150
	without combined indices	---	---	---	---
Total I/O time in msec	with combined indices	20687	2387	17348	13536
	without combined indices	54319	6677	52106	43048
	$\frac{\text{without}}{\text{with}}$	2.620	2.800	3.000	3.180

Table 4.12 Performance of combined indices in a multilist organization. MAXKEY = 5.

Measurements \ Query frequency		F_1	F_2	F_3	F_4
Number of key decodings	with combined indices	1420	179	1392	1166
	without combined indices	2925	367	2869	2386
	$\frac{\text{without}}{\text{with}}$	2.059	2.050	2.096	2.046
Number of data record accessed	with combined indices	1002	111	781	565
	without combined indices	3561	433	3417	2820
	$\frac{\text{without}}{\text{with}}$	3.554	3.946	4.375	4.991
Number of address buckets accessed	with combined indices	1361	171	1330	1115
	without combined indices	---	---	---	---
Total I/O time in msec	with combined indices	20128	23346	17075	13132
	without combined indices	54139	6677	52106	43048
	$\frac{\text{without}}{\text{with}}$	2.690	2.860	3.050	3.280

Table 4.13 Performance of combined indices in a multilist organization. MAXKEY = 6.

Measurements \ Query frequency		F_1	F_2	F_3	F_4
Number of key decodings	with combined indices	1335	168	1314	1107
	without combined indices	2925	367	2869	2386
	without / with	2.191	2.185	2.133	2.169
Number of data record accessed	with combined indices	587	68	488	388
	without combined indices	3561	438	3417	2820
	without / with	6.066	6.441	7.002	7.268
Number of address buckets accessed	with combined indices	1264	161	1259	1054
	without combined indices	---	---	---	---
Total I/O time in msec	with combined indices	14266	1716	12849	10504
	without combined indices	54139	6677	52119	43048
	without / with	3.790	3.890	4.060	4.090

Table 4.14 Performance of combined indices in a multilist organization. MAXKEY = 7.

storage of record identifiers because a combined inverted list consists of the union of several individual lists, by which some duplicate appearances of the same record identifier are eliminated. As a whole, the storage requirements due to the use of combined indices increase or decrease by at most a negligible amount. When combined indices are used in multilist organizations, the query processing is speeded up considerably by retrieving fewer data records. Some update operations are a bit harder to perform while others become easier. Additional storage is necessary for the address buckets of the combined indices. However, this space requirement is limited because the size of each address bucket is small and the total number of address buckets will be small if combined indices are constructed only for the frequently used ones.

Experimental results for heuristically created combined indices do achieve substantial improvement in querying costs for both inverted list and multilist file organizations.

Chapter 5 Clustered File Organizations

In addition to the list structured organizations discussed in the previous two chapters, clustered file organizations are also useful for on-line secondary key retrieval. In terms of the generalized diagram of file organizations and search processes shown in Figure 2.5, a clustered file organization consists of a cluster tree and a set of data records; the former corresponds to the structural file in the diagram and the latter constitutes the data file which contains target records for retrieval. Sections 5.1 and 5.2 of this chapter deal with search methods and structural problems arising in a clustered file organization. Section 5.3 studies the question of choosing an optimal branching ratio for the cluster tree. A model is proposed to compute the querying costs for various branching ratioes by numerical calculations.

5.1 Introduction to Clustered File Organizations

Any query submitted in an on-line retrieval system generally aims at a small subset of the data records in the file. Hence, a file organization used in an on-line system should avoid a linear scan of the whole file. The clustered file organization tackles this problem by partitioning the data file into groups of related records in the hope that the relevant records for any query can be found in a small number of groups. Ideally, it is desirable to classify the records in such a way that items in the same group are always jointly requested by queries. Unfortunately, it is not possible to simultaneously achieve this ideal situation for all queries, because the user population is normally large and volatile. Hence, a realistic goal for a classification is to produce clusters of records such that any query can locate all requested records in a small number of clusters, and that few irrelevant data records are contained in the clusters that are actually examined.

Normally, the similarity between a data record and a query is judged by the respective semantic contents. Any record relevant to a query should bear a semantic content that is similar to the meaning specified in the query. This implies that all data records, which are jointly requested by a query, would also have some semantic indication in common with one another. Therefore, most existing classification algorithms partition records according to their contents --- items with similar content are collated in the same group while items with distinctly different contents are placed into different groups.

As pointed out in section 2.4.2, record contents are often expressed in terms of sets of keywords and/or phrases composed of keywords. The semantic closeness between records is then measurable by a similarity correlation which is a function of the keyword sets corresponding to the two records. A commonly used similarity measure is the cosine function, that is,

$$\cos(q, r) = \frac{(q, r)}{||q||_2 \; ||r||_2}$$

where q and r are the vector representations* of a query and a record, and (q, r) is the inner product of two vectors, and $||\cdot||$ is the ℓ_2 norm of a vector. The classification algorithm then uses a similarity measure to estimate the semantic closeness of records and assigns items of high iter-correlations into the same group. The result of the classification is a set of clusters as shown in Figure 5.1a. Depending on the classification scheme, the same record may or may not be assigned to more than one cluster. Each group of records can in turn be identified by a group profile, or centroid, which is representative of its member records. Murray (1972) proposed several methods for constructing a centroid, or profile. Among them, a widely adopted formulation for a centroid representation is the average of the vector representations of the member records in the cluster. For example, if a cluster

---

* In a file of n distinct keys, each query or record can be conceived of as an n-dimensional vector where the i^{th} component of the vector represents the importance of the i^{th} key for the query or the record in question.

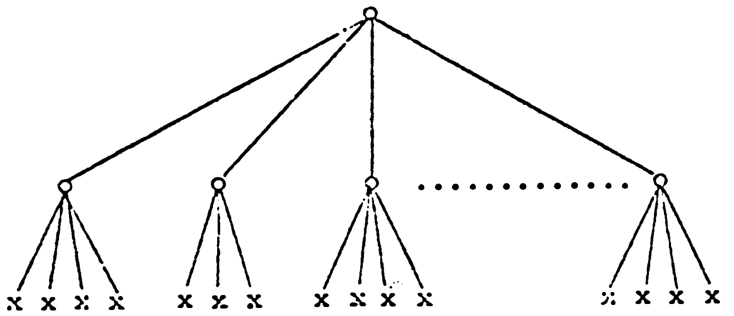
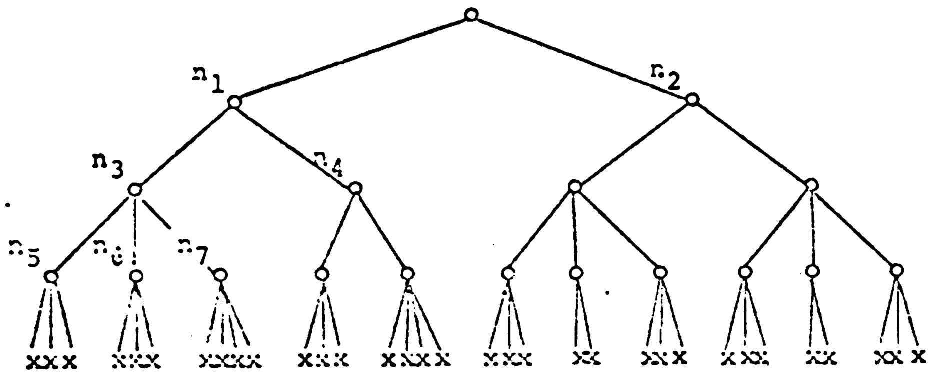


Figure 5.1a A one level clustered file organizations.
(one level of centroid nodes)



o : centroid
x : data record

Figure 5.1b A multi-level clustered file organization contains n records with vector representations $r_1, r_2, \ldots,$ r_n, the centroid representation of the cluster is

$$\frac{1}{n} \sum_{i=1}^{n} r_i.$$

The grouping procedure need not be restricted to the data records alone. The same classification algorithm can be applied for the formation of super groups by grouping the higher level records. that is, the centroids themselves. The grouping procedure can then be applied iteratively to construct still higher level groups. In the end, a multi-level cluster tree is obtained like the one in Figure 5.1b.*

5.2 Some problems in the Search Method and the Structure of Clustered File Organization

.In this section, search methods using clustered file organizations are first briefly introduced. Problems concerning the physical implementation of a cluster tree are discussed next with a view of facilitating the search operation. Finally, certain desirable properties about the structure of a cluster tree --- branching ratio, and balance of tree, etc. --- are investigated.

---

* A multi-level cluster tree need not be always generated in the bottom-up fashion as described in the text. Williamson's method produces multi-level trees in a top-down manner (Williamson 1974).

5.2.1 Search Methods Using a Clustered File Organization

In the case of primary key retrieval, a tree structure --- for example, a binary tree or a Bayer McCreight tree --- is often used for the directory storage needed to obtain the data records. The cluster tree is the counterpart of such directories for secondary key retrieval. Since for each query in a primary key retrieval operation a single data record is required from the file, the directory defines a unique path in the tree for such queries. For secondary key retrieval, more than one data record are normally retrieved for a query. Consequently, the system has to explore an entire subtree of the directory instead of only a single path in the tree.

The subpart of the directory to be examined in a search process is implied by the way in which the cluster tree is generated. By construction, a node in the tree is supposed to indicate the semantic content of the nodes below it, and nodes in close vicinity within the tree are likely to be closely related in semantic contents, and therefore are likely to be jointly retrieved by queries. Thus, a typical search method for a clustered file organization starts at the root of the tree by matching the query with all the top level centroids. A certain number of the highest scoring centroids are then selected. The seauch procedure continues by examining the sons of the selected centroids. This search tactic proceeds successively to the lower levels of the tree until the data records, which are leaves of the tree, are reached.

For example, in the cluster tree of Figure 5.1b, a query would first be matched against the two top level centroids n_1 and n_2. Assuming that node n_1 has a higher correlation with the query than node n_2, the sons of n_1, that is n_3 and n_4, are chosen to be further examined. If only three data records are expected to be retrieved and node n_3 and n_5 correlate more closely with the query than n_4, and n_6,n_7 respectively, then the three data records contained in centroid node n_5 are retrieved.

The number of nodes selected at each level of the tree is dependent on the number of data records which the user intends to retrieve. If a lot of data records are requested by a query, many nodes have to be selected at each level of the tree. Such a search is called a <u>broad search</u>. If, on the other hand, the user is interested in looking at few records only, then a <u>narrow search</u> is conducted in which a small part of the tree is explored. Therefore, in a clustered file organization, the system can flexibly adjust the number of nodes to be retrieved at each level according to the breadth of the retrieval scope.

Some systems allow <u>backtracking</u> in the search, that is, the search can be resumed at some upper level of the tree if the data records previously retrieved turn out to be unsatisfactory for the user. Normally, when a cluster structure is well constructed --- in the sense that records of similar content are included within nodes in close proximity within the tree, and each cluster centroid is indeed representative of the semantic content of its descendant records --- backtracking is rarely necessary in a search process.

5.2.2 Physical Implementation of a Clustered File Organization

A convenient way to store a tree is to reserve two pointers, a sibling pointer and a child pointer, in every internal node of the tree. The sibling pointer of a node points to the next node which has the same parent as the node in question. If the node under consideration is the last son of its parent node, then the sibling pointer points to the parent node. The child pointer in a node points to its first son (if any). Figure 5.2b illustrates such an implementation of the tree in Figure 5.2a.

The same approach with slight modifications can be applied to the storage of the cluster structure. By the definition of a tree, each node, except the root, in a tree has exactly one incoming branch. But this is not necessarily true in a cluster structure. In some applications, it is desirable to allocate a node --- which can be a data record or a centroid --- to more than one cluster if the semantic content of the node is sufficiently close to the contents of more than one cluster.

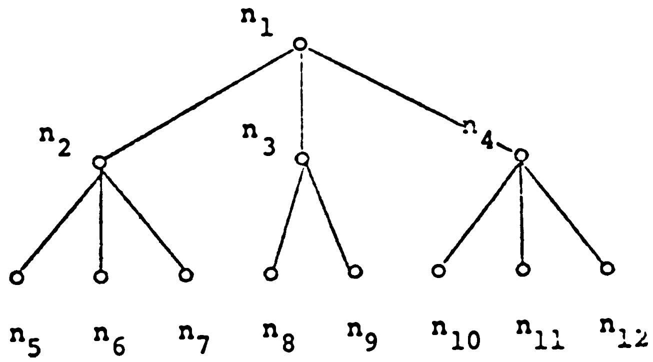


Figure 5.2a A tree structure

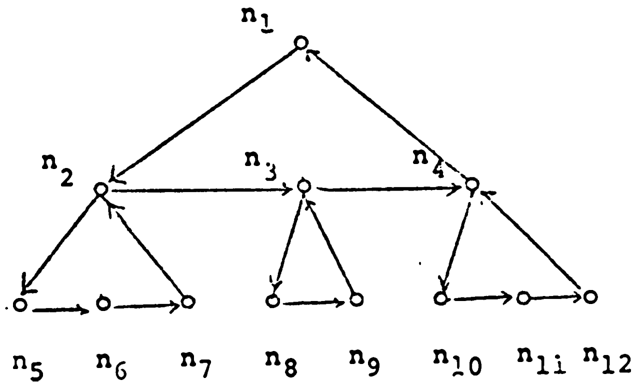


Figure 5.2b Physical implementation of a tree

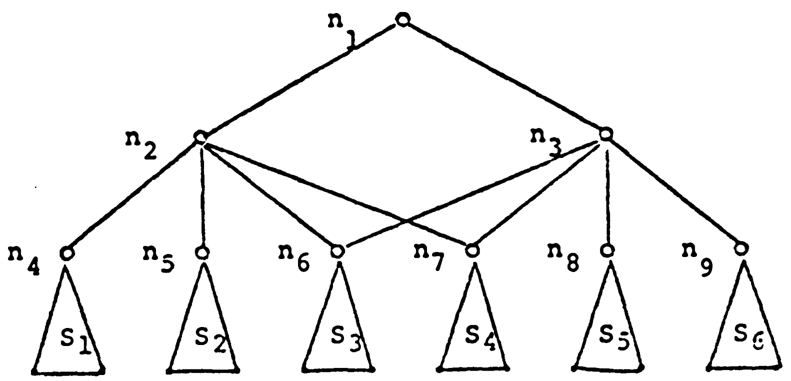
Such cluster structures are said to be <u>non-hierachical</u> clusters.*
Therefore, a non-hierachical cluster structure is not precisely a tree although it has been called a cluster tree in the preceeding paragraphs.

The storage implementation for a hierachical cluster organization is identical with that shown in Figure 5.2b because it is exactly a tree. For non-hierachical cluster organizations, more than one sibling pointer must be reserved in each node which is the son of more than one parent nodes. For example, the cluster structure in Figure 5.3a is a non-hierachical one. Nodes n_6 and n_7 are sons of both n_2 and n_3. In the storage implementation for this cluster structure in Figure 5.3b, two sibling pointers are allocated to node n_6, one representing its relationship with parent node n_2 and the other representing its realtionship with parent node n_3. The same is done for node n_7. In general, for a non-hierachical clustered file organization, each node has as many sibling pointers as the number of parent nodes of which the node is a child.

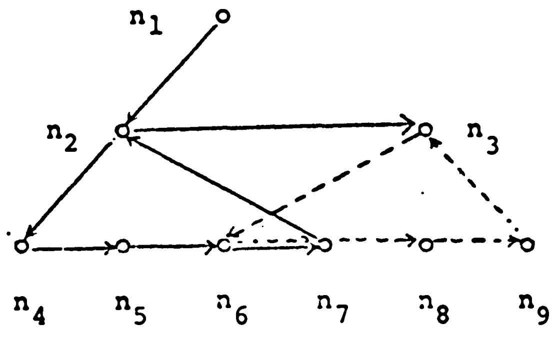
Such implementations of non-hierachical cluster trees are similar to the list linkages in a multilist organization. Consider the cluster tree in Figure 5.3b as an example, nodes n_2, n_4, n_5, n_6, and n_7 can be regarded as a multilist; and nodes

---

* If each data record is allowed to appear in one and only one cluster, the cluster structure is said to be a <u>hierachical</u> cluster.



<u>Figure 5.3a</u> A non-hierachical clustered file organization.



<u>Figure 5.3b</u> Physical implementation of a non-hierachical clustered file organization.

n_3, n_6, n_7, n_8 and n_9 can be considered as records in another multilist. This observation allows one to apply algorithm U of section 3.1.2 --- for non-redundant accesses of records which appear in more than one unordered multilist --- to the search process in a clustered file organization. Assuming that a query would examine only the sons of node n_2 but not sons of node n_3, then the sons of n_2 can be fetched by following the child and sibling pointers. However, if a broader query needs to retrieve the sons for both n_2 and n_3, then one should avoid retrieving centroids n_6 and n_7 repeatedly because they are sons of both n_2 and n_3 (In other words, both n_6 and n_7 appear in two "multilists" simultaneously). Algorithm U of section 3.1.2 can be used precisely in this search process so that n_6 and n_7 are retrieved only once.

A second problem dealing with the storage implementation of a clustered file organization is the actual placement of the centroids and data records in the backup storage devices, say disks or drums. If in the process of searching, a high scoring centroid node is selected for expansion, all of its sons constituting a cluster of lower level centroids or a cluster of data records will be retrieved and matched against the query. Therefore, to allow for fast accesses to all the sons of any node simultaneously, it is desirable to store them in consecutive locations on the disks or drums. But such arrangements are not always possible for non-hierachical clustered organizations. This is because the same node in the cluster tree may appear in several clusters. If the node is not stored redundantly in each cluster to which it belongs, it is theoretically impossible to allot all nodes of every cluster to consecutive locations (Ghosh 1972). The problem is then to pack the nodes so that nodes within the same cluster can be retrieved as fast as possible. This problem will be studied in detail in the next chapter.

5.2.3 Structure of a Cluster Tree

The efficiencies of various tree structures designed for primary key retrieval are strongly influenced by the "shapes" of the trees. For example, a random binary tree may degenerate into a linear list in which case the search procedure becomes a time-consuming linear scan. On the other hand, the path lengths of the right and left subtrees of any node in an AVL tree can differ by at most one (Adel'son-Vel'skii and Landis 1962). Thus, an AVL tree has a much more balanced shape than an arbitrary binary tree; the search time of locating an item is thus guaranteed to be on the order of 1.44 $\log_2 n$ in the worst case.

Another important tree structure for primary key retrieval is the Bayer-McCreight tree of order m, or B-tree in short (Bayer and McCreight 1972). In a B-tree of order m, all leaves of the tree are on the same level; and every node, except for the root, has at least $\lceil \frac{m}{2} \rceil$ sons and at most m sons. The B-tree has thus a balanced tree shape in terms of the node branching ratio --- i.e., the number of sons of a node --- and the path lengths of the tree. The retrieval speed of a B-tree is also very fast --- of order $\log_m n$ --- because of its balanced shape.

For secondary key retrieval, the cluster tree is the counterpart of these trees used for primary key retrieval. It is conceivable that the shape constraints discussed earlier for AVL trees and B-trees may also be useful when applied to cluster tree. The clustering algorithm proposed by Williamson (1974) uses shape constraints similar to those discussed above for B-trees. Williamson's algorithm requires that the number of sons of each node be limited within two bounds. Starting from an empty tree, new data records are assigned to the most closely related cluster. When the size of any cluster reaches the upper limit, the centroid node of the cluster is split into two, and the sons are distributed between the two newly created centroids. The splitting of a centroid node generates one additional son attached to its parent. If this is turn causes an excessive number of sons for this parent, the parent node is split in turn. This procedure propagates upward toward the root of the cluster tree. The cluster tree produced by Williamson's algorithm is therefore similar in shape to a Bayer McCreight tree. All leaves --- data records --- in the cluster tree are on the same level and the branching ratio of every non-leaf node is bounded between two limiting values. Williamson's clustering algorithm is, to the author's knowledge, the first clustering method that tries to keep the tree balanced at all times.

The theoretical worst case running time in generating a cluster tree by Williamson's method is O(n log n) for a file of n data records. The path length in the cluster tree is of the order of log n. Experimentally, Williamson also shows that his clustering algorithm runs substantially faster then most earlier methods.

The above evidence suggest that balanced cluster trees are also improtant for secondary key retrieval. To keep the tree balanced, one requires that the number of sons, i.e. the branching ratio, of each non-leaf node in the tree be approximately equal. The immediate question that arises is the choice of a branching ratio which optimizes the search performance. This problem is studied in detail in the next section.

5.3 Branching Ratio Determination for a Cluster Tree

In this section, the cluster tree is assumed to have a balanced structure, i.e. the branching ratio of every node in the tree is approximately the same. A model is proposed to help the selection of useful branching ratio leading to minimum processing cost.

5.3.1 Branching Ratio and Search Efficiency of a Cluster Tree

The determination of an optimal branching ratio for a cluster tree is similar to the problem of finding an optimal order m for a Bayer-McCreight tree. By definition, a Bayer-McCreight tree requires that the number of sons of each node be between CEIL(m / 2)* and m, and that all leaves of the tree be on the same level (Figure 5.4a). For a fixed number of keys, when the value of m grows, the branching ratio of each node increases, while the height of the tree --- the longest path length from the root to any leaf of the tree --- decreases. This implies that, fewer nodes need to be retrieved during a search because the height of the tree decreases; on the other hand, it becomes more time-consuming to transfer each node from the backup storage device to the main memory because the size of each node grows larger. Bayer and McCreight (1972) formulated an equation for the search casts in terms of the number of nodes retrieved during a key search and the time needed to transfer each node. Numerical calculations of the equation for various values of the variable m produce an optimal value of m for a B-tree.

The shape of a cluster tree is affected by the branching ratio in a similar fashion. When the branching ratio is large,

---

* CEIL(x) = y if y is the smallest integer which is equal to or larger than x.

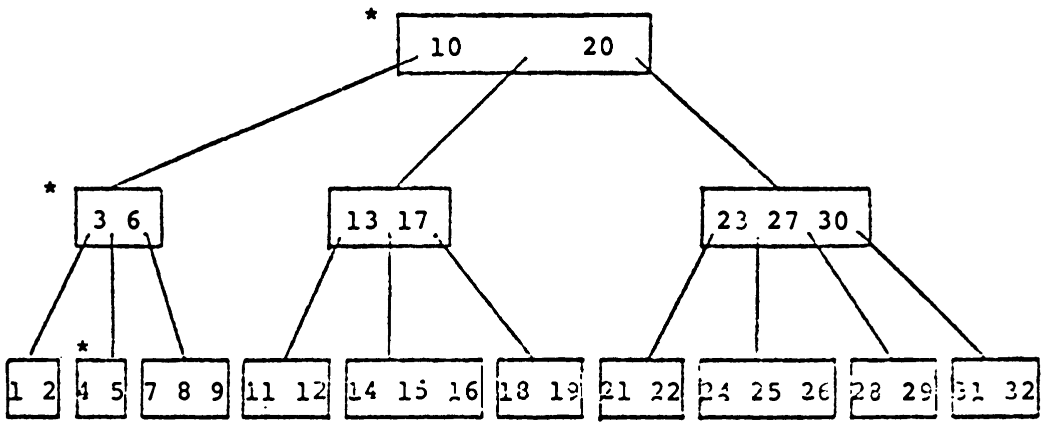


Figure 5.4a A B-tree of order m = 5.

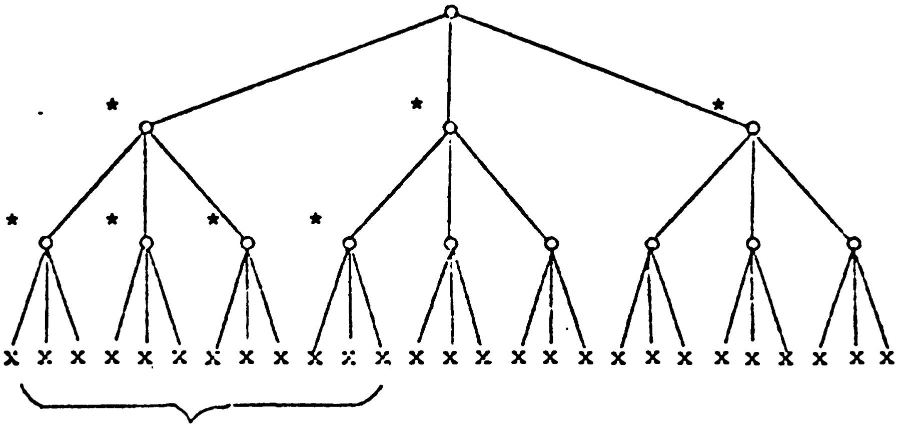


Figure 5.4b A cluster tree with branching ratio equal to 3.

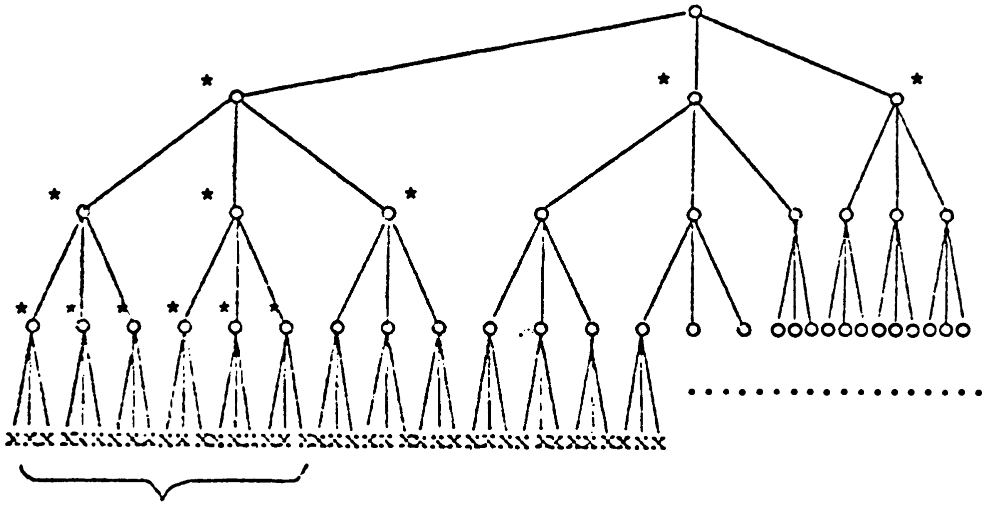
the height of the tree shrinks because each node has more sons. Nevertheless, there is a differentiation must be made between a B-tree and a cluster tree : The B-tree is a directory used for primary key retrieval. Therefore, only certain nodes on a single path of the B-tree are traversed for any query. Referring to the B-tree of Figure 5.4a, if a record identified by key 4 is to be retrieved, only the three nodes marked with asterisks must be examined. These three nodes lie on the same path of the tree. Consider now a secondary key retrieval query requesting the retrieval of twelve relevant records in the clustered file organization of Figure 5.4b. Assuming that the left-most twelve leaves in the tree correspond to the data records that are wanted, then all the higher level centroid nodes marked with asterisks would have to be retrieved and matched before the twelve leaf nodes could be examined. A search query in a clustered file organization generally requests more than one data record; hence, a whole subtree which includes many paths of the cluster tree will be traversed during the search. This fundamental difference renders the optimization for B-trees and for cluster trees dissimilar.

It is also interesting to compare the types of information stored in the nodes of a B-tree and in the nodes of a cluster tree. A node in the B-tree contains a set of ordered keys and a set of pointers. If the key to be searched is not found in a certain node, say node A, of the tree, the next node to be searched is dictated by the relative ordering of the search keys in node A. The information stored in a centroid node of a cluster tree consists of the semantic representation of the contents of the descendant nodes. Generally, the semantic content of a node is represented as a keyword set extracted from the descendant nodes. Thus each node in the cluster tree contains a set of keywords which may be large in number. This implies that the size of each node in a cluster tree is normally large.

In the storage implementation for a cluster tree, the brother nodes attached to the same parent should be blocked into as few physical records as possible since these nodes are always jointly retrieved in the search process. The size of the physical record is often determined by the track size of the device assuming that disks or drums are used.

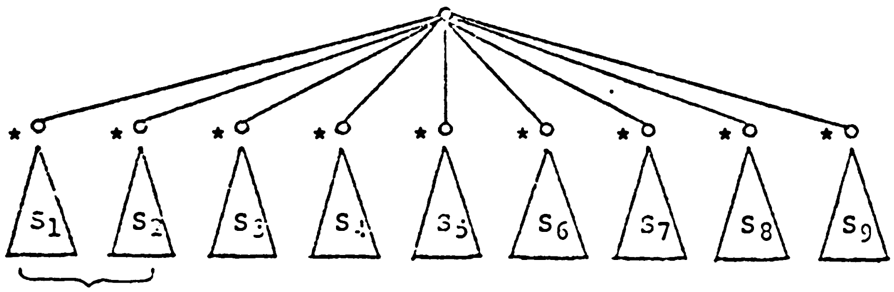
A model for the optimal selection of branching ratios for a cluster tree is proposed in the next subsection. An example is presented in the rest of this subsection to illustrate the search efficiency in a clustered file organization for varying branching ratios. In this example, each node is assumed to be stored in a separate physical record for simplicity.

Example Assume that there are 81 data records included in a file. Figure 5.5a and 5.5b represent the cluster trees for branching ratios 3 and 9 respectively. Consider a query that retrieves the 18 data records specified in the trees. When the cluster tree of branching ratio equal to 3 is used, a total of thirty nodes --- including 12 asterisk centroid nodes and 18



18 data records
to be retrieved

Figure 5.5a A cluster tree with branching ratio equal to 3.



18 data records
to be retrieved

Table 5.5b A cluster tree with branching ratio equal to 9. Each cluster S_i contains 9 data records.

leaves --- must be retrieved. If the branching ratio of the tree increases to 9 shown in Figure 5.5b, all of the nine top level centroid nodes must be retrieved and matched. Then the sons of the two highest scoring centroids are retrieved. Hence a total of 27 nodes are retrieved. In the extreme case when the branching ratio increases to 81, the cluster tree consists of 81 clusters, each of which contains only one data record. Therefore, all of the 81 data records should be retrieved.

This example shows that, as the branching ratio of the cluster tree increases from 3 to 9 and to 81, the number of accesses decreases from 30 to 27 and then increases to 81 when 18 data records are requested by the user. The search cost is thus dependent on the branching ratio of the cluster tree as well as on the number of data records requested by the query. The model shown in the next section formalizes the search costs in a clustered file organization, leading to a study of the effect of the branching ratios on the search efficiencies.

5.3.2 A Search Model for Determining Optimal Branching Ratios for Clustered File Organizations

A Search Model for a Cluster File Organization

The search model for the estimation of optimal branching ratios for a cluster tree can be described in the following aspects :

A. Let N be the total number of data records in a clustered file. The cluster tree is assumed to be a hierarchical one. In the case where non-hierarchical clustered organizations are used, the duplicated occurrences of records are assumed to be negligible --- normally, the redundancy is only a few percent of the file size.

B. The cluster tree is assumed to be balanced in the sense that each node has approximately the same number of sons. To be precise, a balanced cluster tree of branching ratio b for a file of N data records should satisfy the following conditions : For a cluster tree of L levels, the root is at level zero and all the leaves --- that is, data records --- are on the L^{th} level. The path length from the root to any leaf of the tree is the same. The number of nodes on the L^{th} level is N, the total number of data records in the file. The number of nodes at level L - 1 is equal to CEIL (N / b). The average number of sons of each node at level L - 1 is equal to N / CEIL(N / b). The exact number of sons of each node on level L - 1 is restricted to the range between (N / CEIL(N / b)) - δ and (N / CEIL(N / b)) + δ where δ is a small number compared with N / CEIL(N / b). In general, if there are N_i nodes on level i of the cluster tree, the number of nodes on level i - 1 is equal to CEIL (N_i / b). The number of sons of any node at level i - 1 is between (N_i / CEIL(N_i / b)) - δ and (N_i / CEIL(N_i / b)) + δ; and the average number of sons of nodes at level i - 1 is equal to $\overline{S}_{i-1} = N_i / CEIL(N_i / b)$.

The degree to which a cluster tree is balanced is dependent on the size of parameter δ. If δ is large, the number of sons of nodes on the same level may have a wider variation. In general, most clustering algorithms would not produce trees in which all nodes have exactly the same number of sons; nor is it reasonable to require uniformity, because each group of records of similar content may inherently be different in size for any specific file.

These requirements lead to the following definition.

Definition A balanced cluster tree of branching ratio b for a set of N data records satisfies the following requirement :

1) The root of the tree is at level zero. All leaves of the tree are at level L.
2) The average number of sons for nodes on level i is $\overline{S}_i = N_{i+1} / CEIL(N_i / b)$, where $N_L = N$ and $N_j = CEIL(N_{j+1} / b)$ for $j = L - 1, L - 2, \ldots, 0$.
3) The exact number of sons for any node on level i is denoted by S_i and defined as a number in the range $(\overline{S}_i - \delta, \overline{S}_i + \delta)$ where δ is a small constant as compared with any $\overline{S}_i$ for $i = 0, 1, 2, \ldots, L - 1$.

A balanced cluster tree for a branching ratio equal to 3 is illustrated in Figure 5.6.

The above definition for a balanced cluster tree is not

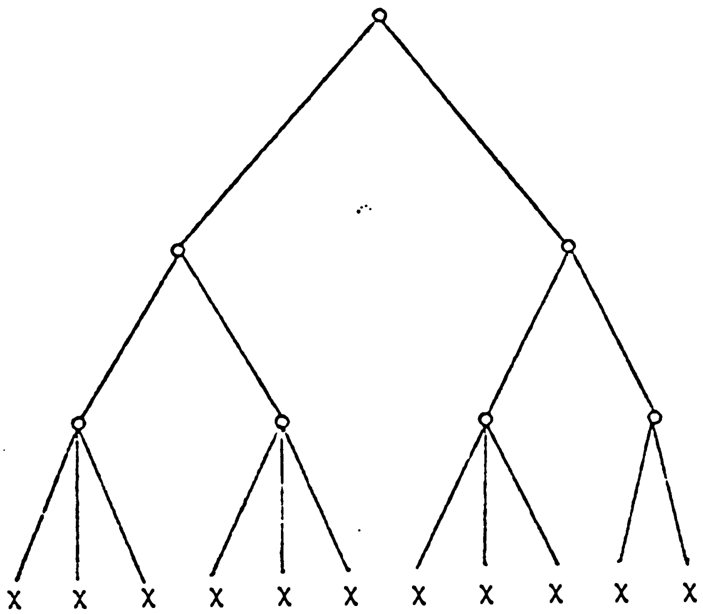


Figure 5.6 A balanced cluster tree of branching ratio equal to three, $\delta = 1$, and N (the total number of data records) = 11.

too stringent to apply to clustered organizations produced by classification algorithms like Williamson's.

C. The search method without backtracking described in section 5.2.1 is assumed in this model. If n data records are requested by a query, these n records are assumed to be closely related in semantic content and are hence located in a "localized subtree" of the cluster tree. For example, if the 5 most closely matched records are requested by a query, these 5 records are located as shown in the configuration of Figure 5.7a instead of the one of Figure 5.7b. Therefore, the n data records that most closely match a given query are assumed to be sons of n_{L-1} (= CEIL(n / $\bar{S}_{L-1}$)) centroid nodes at level L - 1. Thus a total of $n_{L-1}\ \bar{S}_{L-1}$ data records would be retrieved and the highest scoring n records presented to the user. Similarly, these n_{L-1} centroid records are assumed to be sons of n_{L-2} (= CEIL(n_{L-1} / $\bar{S}_{L-2}$) centroid nodes at level L - 2, and $n_{L-2} \cdot S_{L-2}$ nodes at level L - 1 would be retrieved. In general, to retrieve the n most closely related data records for a query, the nodes retrieved at each level of the clustered organization can be summarized as follows :

1) The number of nodes at level i that are retrieved in the search process is equal to $N_i = n_{i-1}\ \bar{S}_{i-1}$ for i = L, L - 1, ..., 1, where n_{L-1} = CEIL(n / $\bar{S}_{L-1}$) and n_j = CEIL (n_{j+1} / $\bar{S}_j$) for j = L - 2, L - 3, ..., 0.

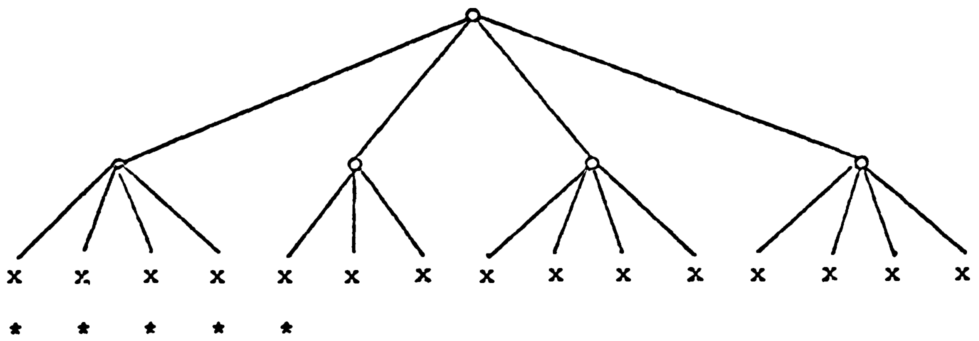


Figure 5.7a A desirable distribution for the five records which are most closely related to a given query.

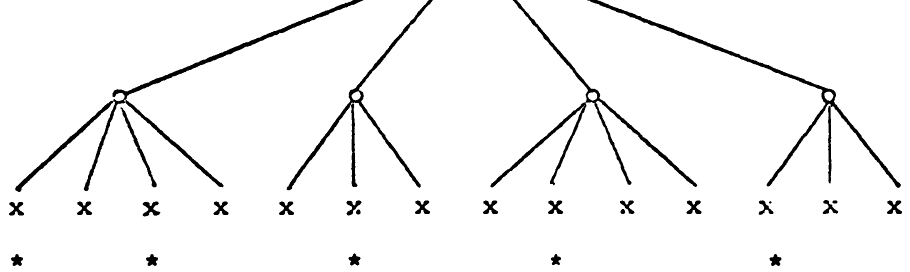


Figure 5.7b An undesirable distribution for the five records which are most closely related to a given query.

2) For all i, the M_i level i nodes specified in 1) are sons of the M_{i-1} level i - 1 nodes specified in 1).

D. The number of bytes required to store each node of the cluster tree must be estimated in order to compute the access time for each node. Assuming that there are an average of k keys in a data record, and that it takes Z bytes to store the information concerning each keyword, the size of a data record, which is a leaf node in the tree, is approximately equal to $k \cdot Z$. Consider a node in the cluster tree which is the centroid of m data records. Let the centroid record be defined as the union of the keys in the m records as described in section 5.1. Then it is reasonable to assume that the number of keys in a centroid record is linearly proportional to the number of data records which are contained in the group of the centroid in question. Thus, if N is the total number of data records in the file and V is the total number of distinct keys in the same file; any centroid record for m data records (for $N \leq m \leq 1$) would contain $f(m) = k + (m - 1) \frac{V-k}{N-1}$ keys. Note that f(m) is a linearly increasing function m and $f(1) = k$ and $f(N) = V$.

Normally, an upper bound, UPLIMIT, is adopted to restrict the number of keys in any centroid record. If the number of keys in a centroid should exceed UPLIMIT, then only UPLIMIT most "important" keys are retained in the centroid record --- some measure should be adopted to estimate the importance of any key with respect to a record, for example, the SMART retrieval system uses the weight associated with each key in a record as a measure of importance.

In conclusion, the number of keys contained in a centroid record consists of m data records is
$KEY(m) = \min \{UPLIMIT, f(m)\}$.

Consider an i^{th} level centroid node in a balanced cluster tree, the number of data records which are descendants of this centroid node is equal to $S_{L-1}\ S_{L-2}\ \ldots\ S_i$. So the number of keys contained in such a centroid record is equal to $KEY(S_{L-1}\ S_{L-2}\ \ldots\ S_i)$. The length of such a centroid record is defined to be $L(i) = \Sigma\ KEY(S_{L-1}\ S_{L-2}\ \ldots\ S_i)$.

Querying Costs Using the Search Model

The cost to process a query contains two components. First, the cost to retrieve the nodes at various levels of the tree; and second, the cost to match the query against these records --- centroid records and/or data records. Since the query-record matches are performed in the central processing unit which is a much faster operation than an I/O access, only the first of the above two components is included in the cost formulation.

Following the conventions of the previous chapters, the cost of making an I/O access to a disk or drum is the sum of the average device latency time T_L and the data transfer time which is proportional to the record length. Normally records are stored as blocks in the backup storage devices. Let B be the block size adopted for the storage of centroid and data records in a clustered file organization.

As specified in paragraph C, the number of level i nodes retrieved in the search process is equal to M_i for i = 1, 2, ..., L. Therefore, the total cost to process a query requesting the n most closely related data records is

$$C_n = \sum_{i=1}^{L} \frac{M_i \, L(i)}{B} \; (T_L + B\, T_t)$$

where L(i) is the average record length for a node on level i of the cluster tree, and T_t is the time required to transfer one byte of data from the backup storage devices to main memory.

As shown in paragraph C, M_i is equal to $n_{i-1}\,\overline{S}_{i-1}$, and n_i is dependent on n because $n_{L-1} = \mathrm{CEIL}(n\ /\ \overline{S}_{i-1})$ and $n_j = \mathrm{CEIL}(n_{j+1}\ /\ \overline{S}_j)$ for j = L - 2, L - 3, ..., 0. Hence C_n is actually a function of n, the number of data records requested by a query. Let P(n) be the probability distribution of variable n. The average search cost for a clustered file organization is then defined as :

$$C = \sum_{n} P(n)\, C_n \; .$$

C is now a function of b because M_i is equal to $n_{i-1}\ \overline{S}_{i-1}$ and $\overline{S}_{i-1}$ (= N_i / CEIL(n_i / b)) is a function of b. Analytical solutions for the optimal values of b which minimize C cannot be obtained easily by equation C. But it is easy to numerically compute equation C for various values of b in order to obtain optimal values for b. The next section describes some results of this like.

5.3.3 Some Numerical Results on Optimal Branching Ratio Determination

An imaginary but reasonable file with the following characteristics is assumed in this example :

N (number of data records in the file) = 5000.

V (number of distinct keys in the file) = 10000.

k (average number of keys per data record) = 70.

Z (average number of bytes to store any key in the data record) = 4.

In the SMART retrieval system, the CRANFIELD aerodynamic collection and the MEDLARS medical collection have 83 and 64 keys per data record on the average (Salton and Yang 1973). So k = 70 is a reasonable choice. Four bytes are used by the SMART system to store information --- concept number and weight of a key (Salton 1968) --- pertaining to each key in a record; the same convention is used in this example.

Suppose that the clustered file is stored on IBM 2314 disk packs. The average latency time, T_L, is equal to 12.5 msec, and

T_t, the time needed to transfer one byte of data, is equal to 0.00322 msec. The size of record blocks, B, is assumed to be the track size of the disk which is 7294 bytes.

The probability distribution P(n) of variable n --- the number of data records requested by a query --- is assumed to be a uniform distribution in the range of (5, 35). Estimations of equation C for various values of b are tabulated in Table 5.1. The result shows that when the branching ratio is in the neighborhood of 16, minimum querying costs are achieved. It is not advisable to have either too large or too small a branching ratio.

The search model in the previous section shows that the cost of query processing in a clustered file organization is a function of (1) the branching ratio of the cluster tree, b; (2) the number of data records to be retrieved by a query, n; (3) the number of data records in the file, N; and (4) the size of the data records, k·Z. It may be interesting to see how the choice of the branching ratio varies with changes in the values of the parameters n, N, or k.

Table 5.2 shows the optimal choices of the branching ratio when the queries retrieve different numbers of data records. Four uniform distributions in the ranges of (5, 35), (25, 55), (45, 75), and (65, 95) are assumed for P(n), the probability density function of n. The average number of data records retrieved by a query, denoted by E(n), is thus 20, 40, 60, or 80 in each case. When E(n) is equal to 20, the average

b	C_{20}	b	C_{20}
2	505	32	163
4	231	34	164
6	172	36	159
8	164	38	159
10	144 *	40	159
12	142 *	42	165
14	157	44	165
16	140 ⊛	46	166
18	151	48	166
20	149	50	192
22	151	52	192
24	154	54	193
26	152	56	193
28	166	58	193
30	166	60	181

N (the total number of data records in the file) = 5000
k (average number of keys per data record) = 70
b :branching ratio
⊛ :optimal branching ratio
* :near optimal branching ratio
C_{20} : query processing cost when the average number of data records retrieved is 20.

Table 5.1 Querying costs and branching ratios for sample file 1 when the average number of records retrieved is equal to 20.

b	C_{20}	C_{40}	C_{60}	C_{80}
2	505	780	1059	1339
4	231	337	443	542
6	172	241	303	375
8	164	210	270	324
10	144 *	185	226	267
12	142 *	180	218	252
14	157	188	225	259
16	140 ⊛	173 ⊛	205 *	238
18	151	181 *	212	243
20	149	177 *	206 *	234 *
22	151	181 *	207	233 *
24	154	181 *	204 *	227 ⊛
26	152	176 *	200 ⊛	228 *
28	166	186	211	241
30	166	187	213	239
32	163	192	228	257
34	164	193	223	253
36	159	190	220	251
38	159	190	214	246
40	159	184	216	240
42	165	190	224	249
44	165	191	226	243
46	166	183	219	[illegible]
48	166	184	220	[illegible]
50	192	202	239	258
52	192	202	240	259
54	193	202	241	261
56	193	193	233	263
58	193	193	234	265
60	181	187	223	254

N (the total number of data records in the file) = 5000.
k (average number of keys per data record) = 70.
b : branching ratio.
C_i : query processing cost when the average number of data records retrieved is i.
⊛ : optimal branching ratio.
* : near optimal branching ratio.

Table 5.2 Querying costs and branching ratios for sample file 1 when various numbers of records are retrieved.

querying cost is minized if the branching ratio is in the range of (10, 16). If E(n) increases to 80, the best branching ratio is approximately in the range of (20, 26). When E(n) is between 20 and 80, e.g. when E(n) is 40 or 60 as shown in Table 5.2, the range of the best branching ratios also lies between the ranges when E(n) is equal to 20 and when E(n) is equal to 80. Therefore, the computational results obtained from the search model confirm the intuitive idea that the best values of the branching ratio are dependent on the number of data records to be retrieved by a query. Furthermore, the more records that are retrieved by the queries, the larger becomes the optimal value of the branching ratio.

When N, the number of data records in the file, is equal to 10,000, the querying costs for various branching ratio are illustrated in Table 5.3. The best values of the branching ratio when n = 10000 are slightly larger than those when N = 5000 previously shown in Table 5.2. The range of the best branching ratio also shifts toward larger values when E(n), the average number of data records retrieved by the queries, grows.

The results when the average number of keys in a data record is reduced to 30 are tabulated in Table 5.4. The values of n and N are the same as those of the file in Table 5.2. Because the size of a physical block in the disk remains unchanged, more data records can be packed into one physical

b	C_{20}	C_{40}	C_{60}	C_{80}
2	506	799	1074	1350
4	227	330	438	537
6	192	261	326	3[illegible]3
8	163	209	263	323
10	164	205	267	283
12	163	201	239	274
14	146 *	176 *	214	243
16	141 ⊛	174 *	207	236
18	156	187	218	249
20	142 ⊛	170 ⊛	199 ⊛	228 ⊛
22	156	187	213	236
24	158	185	208	230 *
26	155	179 *	203 *	232 *
28	157	177 *	202 *	232 *
30	159	180	206	232 *
32	184	213	249	278
34	186	217	245	276
36	181	213	244	274
38	195	217	241	272
40	186	210	243	267
42	187	212	245	270
44	187	213	248	264
46	177	195	230	247
48	177	195	232	240
50	192	191	228	247
52	182	192	230	249
54	182	192	231	250
56	182	182	222	252
58	204	204	245	276
60	204	204	246	278

N (the total number of data records in the file) = 10000.
k (average number of keys perddata record) = 70.
b : branching ratio.
C_i : query processing cost when the average number of data records retrieved is i.
⊛ : optimal branching ratio.
* : near optimal branching ratio.

Table 5.3 Querying costs and branching ratios for sample file 2 when various numbers of records are retrieved.

b	C_{20}	C_{40}	C_{60}	C_{80}
2	456	714	977	1241
4	194	289	384	472
6	137	197	254	311
8	127	165	214	259
10	103	140	173	206
12	109	139	168	195
14	110	133	161	186
16	103	128	152	176
18	95	117	140	162
20	91 *	112	132	152
22	91 *	112	130	148
24	91 *	110	126	141
26	89 ⊛	105 *	121 *	140
28	90 *	103 *	120 *	140
30	91 *	105 *	122 *	139
32	90 *	104 *	122 *	136
34	90 *	105 *	119 *	134
36	90 *	105 *	120 *	135
38	90 *	105 *	117 ⊛	132 *
40	90 *	102 ⊛	118 *	130 *
42	108	120	136	147
44	95	108	125	133
46	96	104 *	121 *	130 *
48	96	105 *	122 *	127 ⊛
50	119	123	142	151
52	119	124	142	151
54	119	124	143	152
56	119	119	139	153
58	119	119	139	154
60	120	120	140	155

N (the total number of data records in the file) = 5000.
k (average number of keys per data record) = 30.
b : branching ratio.
C_i : query processing cost when the average number of data records retrieved is i.
⊛ : optimal branching ratio.
* : near optimal branching ratio.

Table 5.4 Querying costs and branching ratios for sample file 3 when various numbers of records are retrieved.

block (disk track) when the length of the data records decreses. Hence, the optimal values of the branching ratio shown in Table 5.4 are generally larger than those shown in Table 5.2.

In conclusion, numerical results obtained by using the search model show that the optimal branching ratios of cluster trees are dependent on the characteristics of the data base and the queries. Optimal branching ratios for different processing environments seem to indicate that the search model is a reasonable one.

5.4 Summary

The rationale for using a clustered file organization for on-line information systems was reviewed in section 5.1. In section 5.2 the search methods generally used in a clustered file were discussed. Algorithm U of section 3.1.2 was shown to be useful in the search process of a clustered file. The balancing of cluster trees was found to be valuable for the rapid generation and efficient retrieval of records in clustered file organizations. By assuming a balanced cluster tree, a search model was presented in section 5.3 to assess optimal branching ratios of cluster trees. The model was programmed and tested on several sample files. The results showed that neither too high nor too low a value of the branching ratio would be optimal. The best values of the branching ratio are dependent on the number of data records in the file, the number of records retrieved by a query, and the size of the data records.

Chapter 6 Optimal Record Address Allocation Schemes

It was shown earlier that file organization systems consist of two major components : a structural file and a data file. The structural file, if it exists, restricts the search scope for any query to a limited section of the data file; a linear scan of the complete data file is thereby avoided. After searching through the structural file, a subset of records in the data file is normally designated for actual retrieval. The purpose of the structural files is normally twofold : one, to determine the subset of data records to be retrieved using the least amount of work; and two, to eliminate accesses to data records which do not satisfy the query criteria. The material covered in the previous three chapters has dealt primarily with the design of structural files useful for list file and clustered file organizations. In the present chapter, it is assumed that a subset of the data records has already been disignated for retrieval. The problem of interest is then to arrange the records in the data file in such a way that any subset of data records which is to be jointly retrieved by a query can be obtained as fast as possible.

6.1 Introduction to the Consecutive Retrieval Property of Record Address Allocation

Since disk packs are the most widely used devices, data records are assumed to be stored on such devices throughout the discussion. The amount of work needed to retrieve an arbitrary set of records is then dependent on the way in which these records are stored. Ghosh (1972, 1975) treated such record allocation problems by introducing a consecutive retrieval property (CR property) of record addresses. A Q-R system is a set of frequently used queries $Q_1, Q_2, \ldots, Q_m$ together with the sets of data records $R(Q_1), R(Q_2), \ldots, R(Q_m)$ to be retrieved by the queries. A query Q_i and the associated records $R(Q_i)$ are said to possess the consecutive retrieval property if records in set $R(Q_i)$ are assigned sequential addresses in the disk store. Similarly, a Q-R system is said to possess the consecutive retrieval property if each of the m queries possesses the CR property.

If a query Q_i has the CR property, then the set of records $R(Q_i)$ can be retrieved in one access, assuming that the buffer in main memory is large enough to hold all the records included in set $R(Q_i)$. Unfortunately, in many systems, the number of records retrieved by a query is very large and the buffer may not have enough room to simultaneously hold all the records in $R(Q_i)$. Assuming that all records are of the same length and that the buffer holds b records at a time, the $|R(Q_i)|$ * records pertaining to Q_i cannot be retrieved by fewer than

* $|R(Q_i)|$ is defined as the number of elements in the set $R(Q_i)$.

CEIL($|R(Q_i)|$ / b)* accesses even though the records may be stored consecutively on disk. In these circumstances it seems unnecessary to postulate a requirement that all records pertaining to a query be assigned consecutive addresses. Instead, the number of accesses to retrieve the $|R(Q_i)|$ records for Q_i will be minimized so long as these records are stored in at most CEIL($|R(Q_i)|$ / b) blocks of the file. Furthermore, the blocks that contain the record set $R(Q_i)$ need not be assigned to particular localized positions on the disk --- for example, to the tracks on the same cylinder. As pointed out in section 2.5, the time required to perform an I/O access includes a seek time, a latency time, and a data transfer time. If two consecutive accesses to a disk are performed for blocks located on the same cylinder, then the seek time (the time needed to move the disk access arm to the designated cylinder) can be eliminated. But, in a typical multiprogramming computer system, the seek time of a certain device is overlapped with the operations on other disks that may be connected to the same controller (Figure 2.4). The seek time is thus often excluded from the cost accounting for I/O accesses. Secondly, in a data sharing environment where the same file is shared by several concurrent processes, the access commands issued by the same

---

* CEIL(x) = y if and only if y is the smallest integer that is equal to or larger than x.

process may be interwoven with commands from the other concurrent process. Cylinder seeks may thus still occur between two consecutive commands issued by the same process, even though these two commands may be retrieving data blocks from the same cylinder. In summary, to permit the fast retrieval of records pertaining to a query, it is only necessary to assign these records to as few blocks as possible, the exact locations of these blocks need not be consecutive in a multiprogramming data sharing system.

Even when the buffer can hold the complete set $R(Q_i)$ of records, the CR property may not be a useful concept when many queries appear in the system. In fact, the consecutive retrieval property scarcely, if ever, holds simultaneously for all queries in a Q-R system. In fact, as will be shown later, the CR property is not necessarily true even for an arbitrary set of only three queries. Therefore, an alternative solution is to store some records repeatedly in the file with the aim of achieving a <u>CR property with record redundancy</u>. Kou (1974) has proved that, for an arbitrary Q-R system, the problem of arranging the records in such a way that the CR property is obtained with a minimal amount of record redundancy is polynomial complete (Karp 1972). Therefore, it is very hard to arrange the records so that the CR property exists while the redundant storage of records is also minimized at the same time. For most practical purposes, it should then be considered acceptable to obtain the CR property with record redundancy for a Q-R system even though the amount of redundancy is not minimized.

Unfortunately, the introduction of storage redundancy for the records creates difficulties in file maintenance operations. For example, if a record is to be updated, and this record is stored more than once in the file, then all occurrences of this record must be modified. Furthermore, if a file is shared by several concurrent processes, a record which is in the midst of being modified by a certain process must be locked from access by all other processes. Generally, such lockings contribute to a significant amount of system overhead. If a record contains duplicated copies in the file, then an on-line modifications of such a record requires even more system overhead to implement these protective measures. Therefor, redundant occurrences of the same record are not normally recommended for on-line data sharing system.

6.2 Ideal Retrieval Property of Record Address Allocation

The considerations described earlier for the CR property suggest two design guidelines for record address allocation :

a) Records are stored in blocks on disks. The size of each block should be approximately equal to the size of a buffer in the main memory. Data are retrieved from disks in units of blocks.

b) It is not advisable to store more than one copy of any record in an on-line data sharing system.

Based on these two principles, a desirable property of record addresses allocation is defined as follows :

Definition Let b be the number of records that can be stored in a block on the disk*. Let $R(Q_i)$ be the set of records retrieved by query Q_i. Q_i is said to possess the ideal retrieval property (IR property) if the set of records $R(Q_i)$ can be stored in exactly CEIL($|R(Q_i)|$ / b) blocks on the disk storage space. Similarly, a Q-R system of m queries has the IR property if each of the m queries has the IR property.

It was noted previously that the CR property does not often hold for an arbitrary Q-R wywtem. It is interesting to compare the class of Q-R systems which possess the CR property with the class of Q-R systems in which the IR property is true. It will be shown that the latter class "almost" covers the former. The following definition helps illustrate such comparison :

Definition A query Q_i has the near ideal retrieval property (NIR property) if the set of records $R(Q_i)$ pertaining to Q_i is allocated in at most CEIL($|R(Q_i)|$ / b) + 1 blocks. A Q-R system for m queries has the NIR property if each of the m queries either has the IR property or has the NIR property.

Lemma 1 If a Q-R system has the CR property, then it has the NIR property.

---

* For simplicity, each record is assumed to be of the same length throughout this chapter.

<u>proof</u> : The proof is obvious by the definitions of both properties. If a Q-R system has the CR property by a certain ordering of records, say r_1, r_2, $r_3, \ldots$, then the assignment of records r_1, r_2, ..., r_n to the first block, and r_{b+1}, r_{b+2}, ..., r_{2b} to the second block, etc. will cause each query either to have the IR property or to have the NIR property. <u>QED</u>.

<u>Lemma 2</u> A Q-R system satisfying the IR property (hence the NIR property) need not have the CR property.

<u>Proof</u> : This lemma is proved by the following example : Consider a set of three queries Q_1, Q_2 and Q_3. Let each of $R(Q_1)$, $R(Q_2)$ and $R(Q_3)$ consist of the union of four non-empty record sets as shown in Figure 6.1. The number of records contained in each set S_i ($i = 1$, 2, ..., 7) is indicated in the diagram, and $|R(Q_1)| = |R(Q_2)| = 16$ and $|R(Q_3)| = 19$. Suppose that the block size, b, is equal to 10, then CEIL $(|R(Q_i)| \ / \ b) = 2$ for $i = 1, 2, 3$, and query Q_i has the IR property if records in $R(Q_i)$ are assigned to two blocks for each i.

Consider the organization where
records in the set $S_1 \cup S_2 \cup S_3 \cup S_6$ are assigned to block B_1, and
records in the set S_7 are assigned to block B_2, and
records in the set $S_4 \cup S_5$ are assigned to block B_3 (Note that each block contains 10 records). Then the records pertaining to Q_1, Q_2, and Q_3 can be found in blocks B_1, B_3; B_1, B_3;

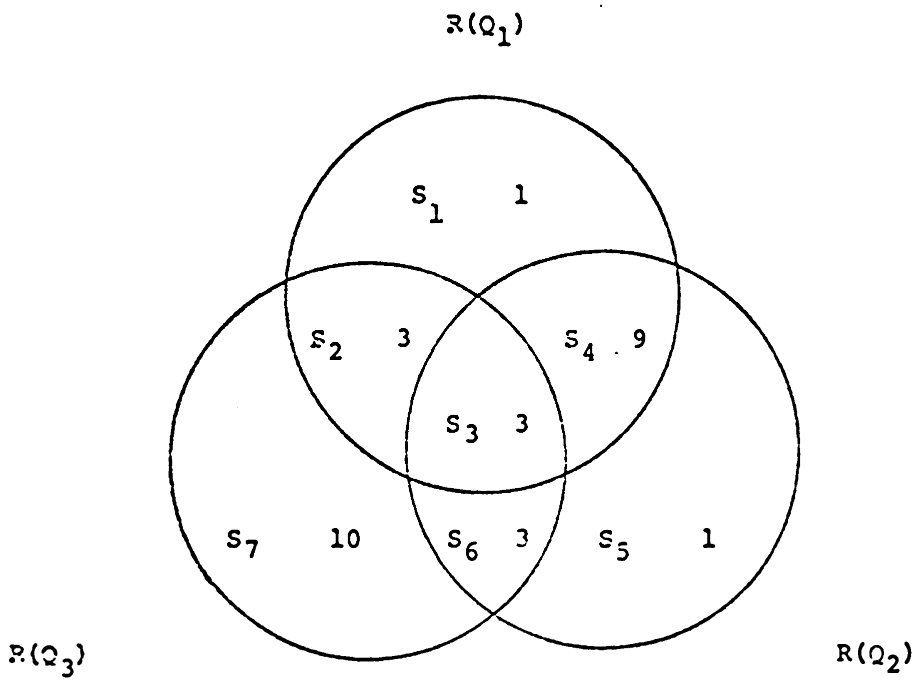


Figure 6.1 A Q-R system of three queries and the associated records.

and B_1, B_2 respectively. Therefore, this Q-R system has the IR property.

However, any attempt to arrange the records can easily convince oneself that this set of three queries does not have the CR property. The formal proof is skipped because it is quite tedious but straightforward. QED.

One consludes from lemmas 1 and 2 that the class of Q-R systems characterized by the CR property is porperly contained in the class of Q-R sytems that have the NIR property. Since the retrieval efficiency of a Q-R system satisfying the NIR property is nearly equivalent to that of a Q-R system satisfying the IR property, one can say that the class of Q-R systems with the IR property "almost" covers the class of Q-R systems with the CR property. Nonetheless, the IR property may not hold for a given Q-R system. In the remainder of this section, a series of results is shown indicating whether the IR property of a Q-R system can be preserved when a new query is added to the system. Intuitively, the easier it is to preserve the IR property of a system under insertion of a new query, the larger will be the class of Q-R systems with IR property. Hence, such results should give some feeling about the likelihood of the existence of the IR property for a given Q-R system.

The first theorem states a sufficient condition for the preservation of the IR property when the pertinent record set $R(Q_0)$ of the newly inserted query Q_0 is a subset of the pertinent record set $R(Q_i)$ of an existing query Q_i. If $|R(Q_0)|$ is very small compared with $|R(Q_i)|$, then the records in $R(Q_0)$ may be scattered among all the blocks which contain the record set $R(Q_i)$. Hence the number of blocks whioh contain record set $R(Q_0)$ may exceed the upper limit --- $\mathrm{CEIL}(|R(Q_0)| / b)$ --- imposed by the definition for the IR property of query Q_0. Formally, the condition is stated as follows :

<u>Theorem 1</u> The IR property for a Q-R system is invariant under the insertion of a new query Q_0 if there exists a query Q_i in the system such that

(a) $R(Q_0)$ is a subset of $R(Q_i)$, and

(b) $|R(Q_0)| \geq [\mathrm{CEIL}(|R(Q_i)| / b) - 1]\, b + 1$

where b is the number of records that can be allocated in a block.

<u>Proof</u> : $|R(Q_0)| \geq [\mathrm{CEIL}(|R(Q_i)| / b) - 1]\, b + 1$ implies that

$$\mathrm{CEIL}(|R(Q_0)| / b) \geq \mathrm{CEIL}(|R(Q_i)| / b) - 1 + \mathrm{CEIL}(1 / b)$$
$$= \mathrm{CEIL}(|R(Q_i)| / b)$$

Since $R(Q_0)$ is a subset of $R(Q_i)$ and Q_i has the IR property, by retrieving the $\mathrm{CEIL}(|R(Q_i)| / b)$ blocks that contain all the records in $R(Q_i)$, one also obtains all the records in $R(Q_0)$. Therefore, records pertinent to Q_0 can be retrieved by no more than the $\mathrm{CEIL}(|R(Q_i)| / b)$ accesses which are less than or equal to CEIL($|R(Q_0)|$ / b) accesses. So, by definition, Q_0 also has the IR property. <u>QED</u>.

Consider next a new query Q_0 whose pertinent records belong to the intersection of the record sets of two existing queries Q_1 and Q_2. For any block, say B_i, which contains at least one record from $R(Q_0)$, the number of records which are pertinent to Q_1 can be as small as $b - Z_1$ (= $|R(Q_1)| - b \cdot$ Floor($|R(Q_1)|$ / b)) and as large as b. Similarly, the same situation is true for Q_2. Thus, the number of records in b_i that are jointly pertinent to Q_1 and Q_2 can be as low as $b - Z_1 - Z_2$ in the worst case. Hence, the IR property is not generally true for the new query Q_2 as shown in the next theorem.

<u>Theorem 2</u> Let Q_1, Q_2, ..., Q_m be the set of queries in a Q-R system which satisfies the IR property. If a new query Q_0 is inserted into the system, and the set of records pertinent to Q_0 is the intersection of the record sets of s existing queries, say Q_1, Q_2, ..., and Q_s, in the system, then the records in $R(Q_0)$ are contained in no more than CEIL($|R(Q_0)|$ / b) + s - 1 blocks.

<u>Proof.</u>: Since Q_i ($1 \leq i \leq m$) has the IR property, record $R(Q_i)$ is allocated to CEIL($|R(Q_i)|$ / b) blocks. Let Z_i be the number of records which are contained in the above CEIL($|R(Q_i)|$

/ b) blocks but do not belong to $R(Q_i)$, then

$$z_i = b \text{ CEIL}(|R(Q_i)| / b) - |R(Q_i)|.$$

Note that $0 \leq z_i \leq b$

Let the records in $R(Q_0)$ be contained in k blocks, say B_1, B_2, ..., B_k. Since $R(Q_0)$ is equal to the intersection of $R(Q_1)$, $R(Q_2)$, ..., and $R(Q_s)$, $R(Q_0)$ is at least as large as

$k\ b - \sum_{i=1}^{s} z_i$, that is

$$|R(Q_0)| \geq k\ b - \sum_{i=1}^{s} z_i \quad \cdots\cdots\cdots\cdots\cdots\cdots\cdots \quad (1)$$

Equality holds in equation (1) when the configuration shown in Figure 6.2 is true.

Since $0 \leq z_i \leq b$ for all i, $0 \leq \sum_{i=1}^{s} z_i \leq s\ b$ is true.

Hence $|R(Q_0)| > k\ b - s\ b$, $|R(Q_0)| / b > k - s$.

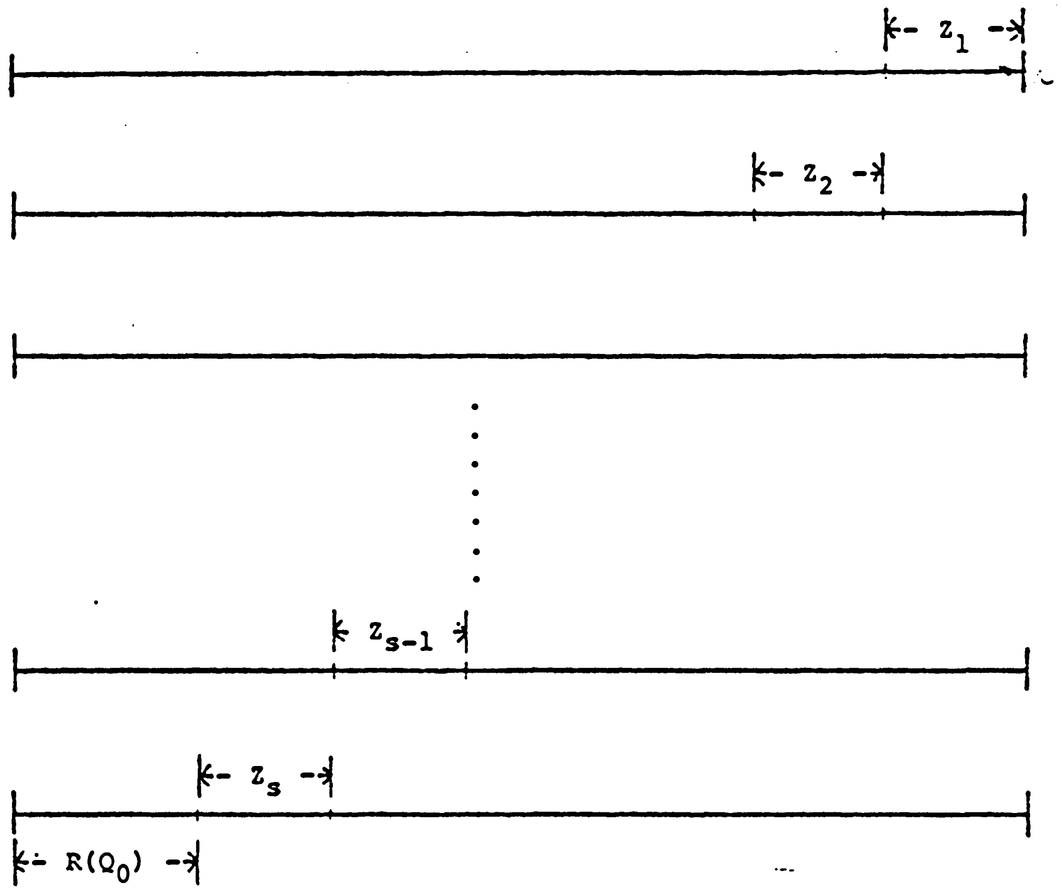
Taking the ceilings of both sides of the equation, one gets

$$\text{CEIL}(|R(Q_0)| / b) \geq k - s + 1 ,$$

i.e. $k \leq \text{CEIL}(|R(Q_0)| / b) + s - 1 \quad \cdots\cdots\cdots\cdots \quad (2)$

where k is the number of blocks which contain the record set $R(Q_0)$. QED.

When s = 2, i.e. when the new query is the intersection



$r_1\ r_2\ r_3$ .... r_{bk}

Records in $R(Q_0)$ are contained in blocks $B_1, B_2, \ldots, B_k$. $r_1, r_2, \ldots, r_{bk}$ is an ordering of the bk records in blocks $B_1, B_2, \ldots, B_k$.

<u>Figure 6.2</u> The configuration for which $|R(Q_0)|$ is minimal for $R(Q_0) = \bigcap_{i=1}^{s} R(Q_i)$.

of two existing queries, the new query is guaranteed to have the "near" IR property. Also, if the values of Z_i are sufficiently small such that $\sum_{i=1}^{s} Z_i < b$ is true, then equation (2) in the above proof can be rewritten as $k \leq CEIL(|R(Q_0)| / b)$, and the new query could have the IR property. This observation is stated in Corollary 2 below :

<u>Corollary 2</u> Let $Q_1, Q_2, \ldots, Q_m$ be the set of queries in a Q-R system which satisfies the IR property. If a new query Q_0 is inserted into the system, and the set of records pertinent to Q_0 is the intersection of the record sets of s existing queries, say $Q_1, Q_2, \ldots, Q_s$, then query Q_0 has the IR property if $\sum_{i=1}^{s} Z_i < b$ is true where $Z_i = b\ CEIL(|R(Q_i)| / b) - |R(Q_i)|$.

If, on the other hand, the record set of a new query Q_0 is the union of the record sets of two existing queries Q_1 and Q_2, the new query may not always have the IR property either. Consider two blocks B_1 and B_2, each of which contains records from $R(Q_0)$. The number of records in $R(Q_0)$ that are contained in B_1 and B_2 can be as small as $b - Z_1 - Z_2$. This is true when B_1 contains $b - Z_1$ records in $R(Q_1)$ but not any record in $R(Q_2)$, and B_2 contains $b - Z_2$ records in $R(Q_2)$ but not any record in $R(Q_1)$. Therefore, arguments similar to those used in the proof of theorem 2 establish the following theorem :

<u>Theorem 3</u> Let $Q_1, Q_2, \ldots, Q_m$ be the set of queries in a C-R system which satisfies the IR property. If a new query Q_0 is inserted to the system, and the set of records pertinent to Q_0 is the union of the record sets of s existing queries, say $Q_1, Q_2, \ldots,$ and Q_s, then the records in $R(Q_0)$ are contained in no more than $\mathrm{CEIL}(|R(Q_0)| / b) + s - 1$ blocks. If the condition $\sum_{i=1}^{s} Z_i < b$ is true, where $Z_i = b\,\mathrm{CEIL}(|R(Q_i)| / b) - |R(Q_i)|$, the records in $R(Q_0)$ are contained in $\mathrm{CEIL}(|R(Q_0)| / b)$ blocks and query Q_0 has the IR property.

<u>Proof</u> : (The proof is very similar to the one for Theorem 2).

<u>Remark</u> These results show that if the record set of a new query consists of the intersection (or union) of the record sets of s existing queries, then the number of blocks in which the records pertaining the new query are contained could be greater than the limit required for the existence of the IR property by as much as s - 1. Such results are however based on a worst case analysis in which Z_i is b - 1 and the record distribution configuration (for example, the one shown in Figure 6.2) is assumed to be the worst possible. Actually, Z_i can be any number between 0 and b - 1; the average value of Z_i is thus equal to (b - 1) / 2. Also, all possible record configurations should be considered to obtain average results. It is conceivable that considerably improved results can be obtained in the average case. In fact, by replacing Z_i by $E(Z_i) = (b - 1) / 2$ in the proofs, the number of blocks to be

retrieved by the new query can be reduced from $CEIL(|R(Q_0)| / b) + s - 1$ to $CEIL(|R(Q_0)| / b) + s / 2 - 1$.

Theorem 4 shows another condition by which the IR property of a Q-R system can be perserved upon insertion of a new query.

Theorem 4 Let $Q_1, Q_2, \ldots, Q_m$ be the set of queries in a Q-R system which has the IR property. After the insertion of a new query Q_0 to the system, the IR property remains true if either

(a) $R(Q_0) \subseteq R(Q_i)$ for $i = 1, 2, \ldots, m$; or

(b) $R(Q_0) \subseteq R(Q_i)$ for $i = 1, 2, \ldots, s$, and

$R(Q_0) \cap R(Q_i) = \phi$ for $i = s+1, s+2, \ldots, m$;

is true.

Proof : The proof of (a) is established first.

Let the records pertaining to Q_0 be included in k blocks. say $B_1, B_2, \ldots, B_k$. Because of condition (a), the k blocks would also be retrieved by each of the m original queries. If $k = CEIL(|R(Q_0)| / b)$, then query Q_0 has the IR property. If $k > CEIL(|R(Q_0)| / b)$, then a repeated application of the procedure SWITCH can rearrange the records so that the records in $R(Q_0)$ are located within $CEIL(|R(Q_0)| / b)$ blocks while, at the same time, the IR properties of the original m queries are preserved.

Procedure SWITCH simply interchanges two records in two

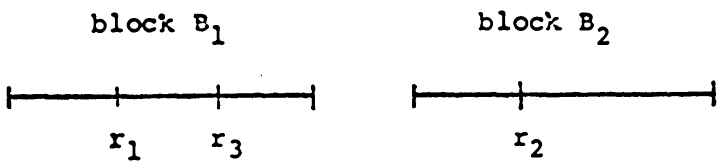


Figure 6.3

separate blocks. Consider the diagram of Figure 6.3. Assuming that records r_1 and r_2 are elements of $R(Q_0)$, but record r_3 is not. Then both blocks B_1 and B_2 need to be retrieved by Q_0 as well as by each of the m queries (because of condition (a)). If r_2 is the only record in $R(Q_0)$ that is contained in B_2, then by interchanging r_2 and r_3, the number of blocks to be retrieved by Q_0 would be reduced by one. At the same time, the IR property for any query Q_i, $i = 1, 2, \ldots, m$, is not destroyed by the switching of r_2 and r_3. Because Q_i had to retrieve B_1 and B_2 before the switch, the number of blocks in which records of $R(Q_i)$ are contained would not be increased by the switch. Hence, if Q_i has the IR property before the switch, it should have the IR property after the switch.

It is obvious how the SWITCH procedure can be applied a finite number of times to obtain the IR property for the new query. This completes the proof of part (a) of the theorem.

Part (b) follows immediately if part (a) is true. QED.

In summary, the IR property is a more realistic view of the record address allocation problem than the CR property. The class of Q-R systems in which the IR property is true "nearly" properly includes the class of systems satisfying the CR property. Unfortunately, many Q-R systems encountered in practical information systems will not possess either the CR or the IR properties. For a Q-R system for which the IR property does not hold, one should then allocate the records in such a manner that the average number of block accesses required to process a query is minimized. This problem is formulated and studied in the next section.

6.3 Minimal Access Retrieval Property of Record Address Allocation

As pointed out in the previous section, if the IR property does not hold for a Q-R system, one should allocate the records so as to minimize the total number of block accesses required to process all the queries in the system. This problem is stated formally through the following definitions.

Definition A Q-R system of m queries Q_1, Q_2, ..., and Q_m and n records r_1, r_2, ..., r_n can be characterized by an access matrix Z and a set of frequencies f_1, f_2, ..., f_m. The access matrix Z is an m x n binary matrix such that $Z_{i,j} = 1$ if and only if query Q_i retrieves record r_j. f_i is the frequency of query Q_i.

Let the records in a Q-R system be assigned to CEIL(n / b) blocks in the file where b is the number of records that can fit into each block*. For a specific record block assignment, the total number of block accesses required to process the queries in the Q-R system is equal to $\sum_{i=1}^{m} a_i$ where a_i is the number of buckets to which the records pertinent to Q_i are assigned.

Definition Any record block assignment for a Q-R system is said to possess the minimal access retrieval property (MAR property) if the total number of block accesses required to process the queries in the system is minimized.

For those Q-R systems in which the IR property is not true, the MAR property is probably the most desirable record assignment criterion as far as the querying efficiency is concerned. In a Q-R system where the IR property is true, a record assignment achieves the MAR property if and only if it achieves the IR property; that is, the two properties are equivalent in such a situation. In this sense, the MAR property is more general than the IR property. Hence, given a Q-R system, one can simply ask the following question : "How can the records be assigned to blocks so that the MAR property is achieved for this system ?". This question is unfortunately

---

* This action is referred to as record block assignment later in the text.

not an easy one to answer. Kirkpatrick (1975) has shown that the organization of records in such a way that MAR property holds is a polynomial complete problem for an arbitrary Q-R system. The time bounds of all known algorithms for polynomial complete problems are exponential functions of the size of the input to the problem. The definition of polynomial completeness and the result proved by Kirkpatrick are given in the rest of this section.

<u>Definition</u> A problem is <u>(non-) deterministic polynomial</u> if there exists a (non-) deterministic Turing machine which solves the problem in polynomial time.

Many difficult problems such as the clique problem, the Hamilton circuit problem, and the travelling salesman problem, etc. are non-deterministically polynomial (Aho, Hopcroft and Ullman 1974); however, it is not known whether they can be solved deterministically in polynomial time. Therefore, the interesting open problem is whether the class of non-deterministically polynomial problems (NP) properly contains the class of deterministically polynomial problems (P) ?
A problem is said to be polynomial complete if a deterministic polynomial solution of this problem can lead to an answer of the question "is P = NP ?". More formally, a polynomial complete problem is defined as follows :

Definition A problem P_1 is polynomial complete if and only if the following conditions are satisfied :

(a) The existence of a deterministic polynomial algorithm for P_1 implies that P = NP is true.

(b) If P = NP is true, then there exists a deterministic polynomial algorithm for P_1.

An answer to the question "is P = NP ?" has not been found so far. Therefore, if a problem is polynomial complete, it is not likely that a deterministic polynomial algorithm is found for it, and one must resort to an exponential algorithm. The next theorem shows that the construction of the MAR property for a Q-R system is polynomial complete. The proof is performed by reducing the clique problem --- which is polynomial complete --- to the problem at hand. A k-clique in a graph G is a complete subgraph (every pair of distinct vertices in the subgraph is connected by an edge) of G with k vetices. The clique problem may be expressed as follows : Given a graph of n vertices and an integer k such that $\frac{n}{3} \leq k < \frac{n}{2}$, does G contain a k-clique ?

Theorem 5 Given an arbitrary Q-R system, the problem of finding a record block assignment with the MAR property is polynomial complete.

Proof : The proof consists in showing that every clique problem is polynomially transformable to a record block assignment problem for a Q-R system for which all query frequencies are identical, and that the records are assigned to two blocks only.

Let G be an arbitrary graph with vertices $V_1, V_2, \ldots, V_n$, and edges $e_1, e_2, \ldots, e_m$. Suppose that one wants to know if G has a k-clique where $\frac{n}{3} \leq k < \frac{n}{2}$. One can construct an m by 2n - 2k binary matrix Z where

$$z_{i,j} = \begin{cases} 1 & \text{if } j > n \text{ or if } V_j \text{ is incident on } e_i, \\ 0 & \text{otherwise.} \end{cases}$$

The first n columns of Z constitute a matrix representation of graph G. Every element in the last n - 2k columns of Z is assigned the value 1 (Figure 6.4). Z can be considered as the access matrix for a Q-R system of m queries and 2n - 2k records. Let query Q_i correspond to the i^{th} row of Z and record r_j correspond to the j^{th} column of Z. Note that each of the records $r_{n+1}, r_{n+2}, \ldots, r_{2n-2k}$ are relevant to all m queries. The record assignment problem for this Q-R system is to assign the 2n - 2k records to 2 blocks --- each contains n - k records --- such that the total number of block accesses denoted by A, for the m queries is minimized. Note that A is at most 2m. The arguments in paragraphs (a) and (b) below prove that G has a k-clique if and only if A is less than 2m - k (k + 1) / 2. Thus, if there is a deterministic algorithm which solves the record assignment in polynomial time, one can solve the clique problem in polynomial time. Therefore,

$$
Z = \begin{array}{c} \\ e_1 \\ e_2 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ e_m \end{array}
\begin{array}{c}
\begin{array}{cccccc} v_1 & v_2 \cdots\cdots & v_n & v_{n+1} & \cdots\cdots & v_{2n-2k} \end{array} \\
\left(\begin{array}{cccccc}
 & & & 1 & 1 \cdots\cdots\cdots\cdots & 1 \\
 & & & 1 & 1 \cdots\cdots\cdots\cdots & 1 \\
 & & & & & \\
 & & & & & \\
 & & & & & \\
 & & & & & \\
 & & & 1 & 1 \cdots\cdots\cdots\cdots & 1
\end{array}\right)
\end{array}
$$

$$\Updownarrow$$

$$
\begin{array}{c} \\ Q_1 \\ Q_2 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ Q_m \end{array}
\begin{array}{c}
\begin{array}{cccccc} r_1 & r_2 \cdots\cdots & r_n & r_{n+1} & \cdots\cdots & r_{2n-2k} \end{array} \\
\left(\begin{array}{cccccc}
 & & & 1 & 1 \cdots\cdots\cdots\cdots & 1 \\
 & & & 1 & 1 \cdots\cdots\cdots\cdots & 1 \\
 & & & & & \\
 & & & & & \\
 & & & & & \\
 & & & & & \\
 & & & 1 & 1 \cdots\cdots\cdots\cdots & 1
\end{array}\right)
\end{array}
$$

<u>Figure 6.4</u> Matrix Z for a clique problem and the access for a corresponding record assignment problem.

the record assignment problem is polynomial complete because the clique problem is polynomial complete.

(a) "If there is a k-clique in G, then $A < 2m - k(k + 1) / 2$."

Let $V_{n-k+1}, V_{n-k+2}, \ldots, V_n$ be the vertices in the k-clique. Let $e_1, e_2, \ldots, e_{k(k+1)/2}$ be the edges that constitute the k-clique. Then, by assigning $\{r_{n-k+1}, r_{n-k+2}, \ldots, r_n, r_{n+1}, \ldots, r_{2n-2k}\}$ to block B_1 and $\{r_1, r_2, \ldots, r_{n-k}\}$ to block B_2, queries $Q_1, Q_2, \ldots, Q_{k(k+1)/2}$ need to retrieve block B_1 but not block B_2 --- because edges $e_1, e_2, \ldots, e_{k(k+1)/2}$ touch vertices $\{V_{n-k+1}, V_{n-k+2}, \ldots, V_n\}$ only. So, A, the total number of block accesses, is less than $2m - k(k + 1) / 2$.

(b) "If $A < 2m - k(k + 1) / 2$, then there is a k-clique in G."

Let B_1 and B_2 be the two blocks of records. All the records $r_{n+1}, r_{n+2}, \ldots, r_{2n-2k}$ should be jointly assigned to the same block, because if this were not the case A would have to be equal to 2m because each of these records is relevant to all the m queries. Let these records be assigned to block B_2 say; then every query must retrieve B_2. Since $A < 2m - k(k + 1) / 2$, at least $k(k + 1) / 2$ queries do not access any of the records in block B_1. This implies that at least $k(k + 1) / 2$ edges do not touch any of the $n - k$ vertices whose corresponding records are assigned to B_1. Hence, these edges must be incident upon the rest of the k vertices in G. But there can be at most k (k + 1) / 2 edges that are incident to k vertices; therefore, these k vertices should form a k-clique. QED.

6.4 A Heuristic Algorithm for the Minimal Access Retrieval Property of Record Address Allocation

Because the record block assignment problem is polynomial complete, it is clear that the design of a polynomial algorithm for the problem is very difficult, if not impossible. An exponential algorithm is bound to be unrealistic because of the large number of records typically encountered in an information system. In this section, a heuristic algorithm is proposed for the assignment of records to blocks. Experimental results are obtained to test the usefulness of the heuristic algorithm.

6.4.1 The Heuristic Algorithm

The heuristic algorithm is similar to the one (Algorithm II of section 4.3.1) used to create the combined indices for an inverted list organization. The algorithm consists of two phases. Phase one classifies the records into groups of bounded size such that the records in the same group are often jointly retrieved by the queries. Phase two is a more refined procedure which assigns records in the same group to several blocks. Before formally describing the algorithm, a few definitions are useful.

<u>Definition</u> An <u>access matrix</u> Z is an m x n binary matrix such that $Z_{i,j} = 1$ if and only if query Q_i retrieves record r_j. The j^{th} column of Z, denoted by Z(j), is called the <u>access vector</u> of record r_j. Z(j) specifies the queries which retrieve record r_j. The access vector of a block, B, consisting of s record, r_1, r_2, ..., and r_s, is defined as the logical OR of the access vectors of the s records. That is, $Z(B) = Z(1) \vee Z(2) \vee \ldots \vee Z(s)$.

The following example illustrates the rationale of the heuristic algorithm :

<u>Example</u> To minimize the number of blocks accessed by a query, the record assignment of a specific block can be considered ideal for a query if either (a) this query is relevant to every record assigned to the specific block, or (b) this query is relevant to none of the records in the block. Criterion (a) is probably more important than criterion (b) because there generally exist much more irrelevant records for a query than relevant ones. Consider a Q-R system of four queries Q_1, Q_2, Q_3 and Q_4. Suppose that a certain block B has the access vector $Z(B) = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$. Let record r_i have the access vector

$Z(i) = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}$. Consider the assignment of record r_i to block B.

This assignment is "advantageous" for queries Q_1 and Q_4 because of criteria (b) and (a) respectively. But the assignment is potentially disadvantageous for queries Q_2 and Q_3 for the same reasons.

The above example leads to the following definitions.

Definition The degree of match concerning the assignment of record r_i to block B, denoted by $M(B, r_i)$, is defined as $\sum_{i=1}^{\ell} f_i$ where $Q_1, Q_2, \ldots,$ and Q_ℓ are all the queries which retrieve both block B and record r_i; and f_i is the frequency of query Q_i.

Definition The degree of mismatch concerning the assignment of record r_i to block B, denoted by $U(B, r_i)$, is defined as $\sum_{i=1}^{\ell} f_i$ if, before r_i is assigned to B, $Q_1, Q_2, \ldots,$ and Q_ℓ are all the queries which retrieve either B or r_i, but not both.

The heuristic algorithm to be described creates one block at a time. In the process of assigning records to the block under construction, the particular record which produces the maximum match with the block is selected for assignment. If more than one record has the same maximum match with the block, then the one with the minimum mismatch is selected.

Algorithm R (Record Block Assignment) Given a Q-R system of m queries and n records with access matrix Z, assign b records to each block in the file.

Phase One Use the GROTRE clustering algorithm (Williamson 1974) to produce groups of records $G_1, G_2, \ldots, G_g$ where the number of records in each group is bounded between two limiting constants S_1 and S_2.

Phase Two Applying the following steps to each of the groups G_a, $a = 1, 2, \ldots, g$:

Step 1 : If G_a is empty then STOP.

(Steps 2 through 5 create a block B)

Step 2 : (initialize the block B) Assign to B an arbitrary record r_c of group G_a. $G_a \leftarrow G_a - \{r_c\}$.

Step 3 : Repeat step 4 for $i = 1, 2, \ldots, b - 1$. (At each iteration, assign to B a record from G_a which produces the maximum degree of match)

Step 4 : Select a record r_{b_i} from G_a such that $M(B, r_{b_i}) = \max\{M(B, r_j) \mid r_j \in G_a\}$. If more than one record satisfies the above condition, select the record r_{b_i} that has minimum degree of mismatch, $U(B, r_j)$. Assign r_{b_i} to B, $G_a \leftarrow G_a - \{r_{b_i}\}$.

Step 5 : Output block B.

Step 6 : $B \leftarrow$ empty set. Go to step 1.

Algorithm R is completely parallel to Algorithm II of section 4.3.1. Therefore, the same running time, $O(n \log n)$, also applies to Algorithm R.

6.4.2 Experimental Results

Algorithm R is tested on a randomly generated Q-R system containing 55 queries and 400 records. The number of records pertinent to each query is uniformly distributed in the range between 20 and 40. Therefore, an average of 30 records are retrieved by each query. The 400 records are generated as ten groups. It is more likely that records in the same group are to be jointly retrieved by the queries then records in different groups.

Algorithm R is then applied to the above Q-R system. Phase one classifies the 400 records into ten clusters as shown in Table 6.1. Phase two of Algorithm R assigns the records in each cluster to blocks of size b. A uniform query frequency distribution in the range of (5, 10) denoted by F_1, is used in the Algorithm.

In the experiments, five values, 5, 10, 15, 20, and 40 are assumed for parameter b. The system performance for records assigned to blocks according to algorithm R is compared with the performance for records that are randomly assigned to blocks.

Table 6.2 shows the number of block accesses required to process the 55 queries when the frequencies of the queries are distributed according to F_1. For most values of b, the

cluster	number of records in the cluster
1	46
2	40
3	49
4	43
5	39
6	43
7	37
8	31
9	37
10	35

Table 6.1 Numbers of records in the clusters of the test file.

number of accesses is at most half when the records are assigned by way of Algorithm R compared with a random assignment.

Three additional frequency distributions F_2, F_3 and F_4 are used to test the effectiveness of the record block assignment when the query frequencies alter. The three distributions are also uniform distributions. F_2 and F_4 have frequency ranges (1, 1) and (1, 10) respectively. F_3 has the same range as F_1. The frequency assignment for each of the 55 queries is however different in F_1 and F_3. The effectiveness of the system when the query frequency distribution changes from F_1 to F_2, F_3, or F_4 is shown in Tables 6.3, 6.4, and 6.5. In each case, there are only minor deteriorations in effectiveness.

record block assignment \ block size b	5	10	15	20	40
(a) random assignment	11632	9833	8183	7079	4208
(b) assignment by Algorithm R	5726	4478	3941	3528	2426
$\frac{(a)}{(b)}$	2.03	2.20	2.08	2.01	1.73

Table 6.2 The number of block accesses required to process the queries when the records are randomly organized as compared with the organization by Algorithm R. Query frequency distribution is F_1.

record block assignment \ block size b	5	10	15	20	40
(a) random assignment	1453	1225	1018	880	525
(b) assignment by Algorithm R	721	564	496	444	304
$\frac{(a)}{(b)}$	2.02	2.17	2.05	1.98	1.73

Table 6.3 The number of block accesses required to process the queries when the records are randomly organized as compared with the organization by Algorithm R. Query frequency distribution is F_2.

block size b / record block assignment	5	10	15	20	40
(a) random assignment	11275	9495	7898	6833	4090
(b) assignment by Algorithm R	5599	4379	3856	3461	2378
$\frac{(a)}{(b)}$	2.01	2.17	2.05	1.97	1.72

Table 6.4 The number of block accesses required to process the queries when the records are randomly organized as compared with the organization by Algorithm R. Query frequency distribution is F_3.

block size b / record block assignment	5	10	15	20	40
(a) random assignment	9771	8268	6844	5901	3519
(b) assignment by Algorithm R	4867	3808	3352	3013	2045
$\frac{(a)}{(b)}$	2.01	2.17	2.04	1.96	1.72

Table 6.5 The number of block accesses required to process the queries when the records are randomly organized as compared with the organization by Algorithm R. Query frequency distribution is F_4.

Chapter 7 Summary and Further Research

A new class of hybrid list organization is introduced in this thesis. Each file element in this class is determined by the value assigned to a list length parameter ℓ_{th}. Both inverted list and multilist organizations are shown to be particular file elements in this class, and neither of them normally represents the optimal choice in terms of querying and updating costs among all file elements in the class. This result is shown to be true by using either a simplified analytical model and a more complicated simulation model. Although the simulation models seem to reflect the characteristics of information systems fairly closely, it would be interesting to further confirm such results by conducting experiments on actual data bases and actual user queries. Also, it is argued that, when the list lengths change due to update operations, the conversion of a lengthened multilist into an inverted list (or vice versa) can be postponed until file reorganization time without seriously degrading system performances. Experiments on real data bases and queries can also provide evidence to support this observation. On the other hand, it would be challenging to obtain analytical results for complicated system models. This can be a very difficult task if one considers the large number of parameters required to describe a realistic system configuration.

A new method is suggested for the creation of combined indices for list structure files. Instead of creating all possible combined indices, only the frequently used ones are included. Such an approach can be useful for systems in which a small subset of the keys is used much more frequently than the remaining keys. The combined indices are shown to be valuable in reducing querying costs without causing difficulties in update operations and storage requirements. Compared with the previous approaches by Lum (1970) and Wong and Chiang (1971), the method described in this thsis does not achieve the same large improvements in query costs. However the horrendous updating costs and/or storage requiraments implicit in their methods are avoided. A future work on combined indices is the design of new methods which simultaneously ascertain the best querying and updating performances while avoiding excessive storage requirements.

The search methods and structural implementation of a clustered file is discussed in chapter 5. The balancing of of cluster trees is shown to be valuable both for cluster generation and for cluster searching. A model is established in order to estimate the optimal branching ratios for balanced cluster trees. The results show that the optimal branching ratio is in fact affected by the data base and query characteristics. However, this model takes into account only one aspect of clustered file evaluation --- namely the best choice of the branching ratio. The performance of a clustered file is also dependent upon other factors, for example, the degree of inter-cluster overlapping, various methods of query searches, etc.. Furthermore, the balancing concept adopted in this thesis as well as in Williamson's work (Williamson 1974) is stringent one in that all leaves in the tree must have the same path length. If a controlled variation of the path lengths is allowed in a cluster tree, it is conceivable that the generation of cluster trees would be less time-consuming and the updating less costly, while the querying cost is not greatly affected. Therefore, the evaluation model should contain simultaneously the aspects of inter-cluster overlapping, search methods, branching ratio, and degree of balancing, etc.. Further work along this line is both interesting and urgent.

The last part of the thesis assumes that a subset of the data records has already been designated for retrieval by using the structural files. The problem is then to assign the data records to blocks in the disk-like storage devices so as to minimize the average number of I/O accesses required to process a query. The problem was formulated as a Minimal Access Retrieval Problem and is shown to be polynomial complete. Certain heuristic methods prove to be useful in reducing the I/O accesses. It is not clear yet how close to optimal these heuristic methods are. Some further work is required to show that these methods are close enough to optimal or to propose better heuristics.

Appendix 1 Random Variable Generation by Computers

1. Procedure to generate random variates of a uniform distribution.

   The built-in function RAND( ) in PL/1 generates random variates in the range of 0 and 1.

2. Procedure to generate random variates of a beta distribution, $\beta(a,b)$, for integral parameters a and b.

   Step 1 : Use RAND( ) to generate $a + b - 1$ random members $u_1, u_2, \ldots, u_{a+b-1}$.

   Step 2 : Sort $u_1, u_2, \ldots,$ and u_{a+b-1}; renumber then so that $u_{i_1} \leq u_{i_2} \leq \cdots \leq u_{i_{a+b-1}}$.

   Step 3 : Output $x = u_{i_s}$.

3. Procedure to generate an exponential distribution with mean u.

   Step 1 : Use RAND( ) to generate a random number u.

   Step 2 : Output $x = - \mu \log u$.

More detailed analyses of these procedures can be fpund in Fishman (1973).

REFERENCES

[1] Adel'son-Vel'skii, G.M. and Landis, Y.M. (1962), "An algorithm for the organization of information", Dokl. Akad. Nauk SSUR 146, pp 263-266.

[2] Aho, A., Hopcroft, J.E. and Ullman, J.D. (1974), The design and analysis of computer algorithm, Addison-Wesley Publishing Company, 1974.

[3] Bayer, R. and McCreight, E. (1972), "Organization and maintenance of large ordered indexes", Acta Informatica 1, pp 173-189.

[4] Bonner, R.E. (1964), "On some clustering Techniques", IBM Journal of Research and Development 8,1, January 1964.

[5] CODASYL (1971), CODASYL data base task group report, ACM. N.Y. April 1971.

[6] Codd, E.F. (1971), " Date base sublanguage founded on the relational calculus", Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, ACM. N. Y. 1971.

[7] Date, C.J. (1975), An introduction to data base systems, Addison-Wesley 1975.

[8] Dattola, R.T. (1973), "Automatic classification in document retrieval systems", Ph.D. thesis, Cornell University, Computer Science Department, May 1973.

[9] Files, J. and Huskey, H. (1969), "An information retrieval system based on superimposed coding", AFIP Vol. 35, 1969.

[10] Fishman, G.S. (1973), Concepts and methods in discrete event digital simulation, John Wiley and Sons, 1973.

[11] Ghosh, S. (1972), "File organization : The consecutive retrieval property", CACM 15, 9, September 1972.

[12] Ghosh, S. (1975), "Consecutive storage of relevant records with redundancy", CACM 18, 8, August 1975.

[13] Gustafson, R. (1969), "A randomized combinotorial file structure for storage and retrieval systems", Ph.D. thesis, University of South Carolina 1969.

[14] Hsiao, D. and Harary, F. (1970), "A formal system for information retrieval from files", CACM 13, 2, February 1970, pp 67-73.

[15] IBM (1973), IMS/360 Version 2 System/Application Design Guide, IBM Corp. SH20-0910-4, September 1973.

[16] Karp, R.M. (1972), "Reducibility among combinational problems", In Complexity of Computer Computation, R.E. Miller and J.W. Thatcher (eds.) Plenum Press, New York 1972, pp 85-104.

[17] Kirkpatrick, D. (1975), Personal communication.

[18] Knuth, D.E. (1973), The Art of Computer Programming, Vol. 3, Searching and Sorting, Addison-Wesley Publishing Company, Reading, Mass., 1973.

[19] Kou, L.T. (1974), "Polynomial complete consecutive information retrieval problem with nonredundancy", TR 74-193 Cornell University, Department of Computer Science, January 1974.

[20] Lancaster, F.W. and Fayen, E.G. (1973), Information Retrieval On-line, John Wiley & Sons, Inc. 1973.

[21] Lefkovitz, D. (1969), File Structures for On-line Systems, Spartan Books, New York, 1969.

[22] Lefkovitz, D. (1974), Data Management for On-line Systems, Hayden Book Company, Inc., 1974.

[23] Litofsky, B. (1969), "Utility of automatic classification systems for information storage and retrieval", Ph.D. thesis, University of Pennsylvania, 1969.

[24] Lum, V.Y. (1970), "Multi-attribute retrieval with combined indices", CACM 13, 11, November 1970, pp 660-665.

[25] Lum, V., Yuen, D. and Dodd, M. (1971), "Key-to-address transformation techniques, a fundamental performance study on large existing formatted files", CACM 14, 4, April 1971.

[26] Morris, R. (1968), "Scatter storage techniques", CACM 11, 1, January 1968.

[27] Murray, D.M. (1972), "Document retrieval based on clustered files", Ph.D. thesis, Cornell University, Department of Computer Science, June 1972.

[28] Olle, T.W. (1969), "Data structures and storage structures for generalized file processing", Proc. of the File 68 International Seminar on File Organization, Copenhagen, 1968, pp 285-294.

[29] Pearson, K. (1968), Tables of the Incomplete Beta Function, Cambridge, Biometrika 1968.

[30] Rivest, R. (1974), "Self organizing sequential file", 15th Annual Symposium on Switching and Automata Theory, October 1974.

[31] Salton, G. (1968), Automatic Information Organization and Retrieval, McGraw-Hill Book Co., N. Y. 1968.

[32] Salton, G. (1971), The SMART Retrieval System - Experiments in Automatic Document Processing, Prentice-Hall Inc., Englewood Cliffs, N. J. 1971.

[33] Salton, G. (1974), "A theory of indexing", Technique Report TR 74-203, Department of Computer Science, Cornell University, March 1974.

[34] Salton, G. (1975), Dynamic Information and Library Processing, Prentice-Hall Inc., 1975.

[35] Salton, G. and Yang, C.S. (1973), "On the specification of term values in automatic indexing", The Journal of Documentation 29, 4, December 1973.

[36] Salton, G., Yang, C.S. and Yu, C.T. (1974), "Contribution to the theory of index", Proc. IFIP Congress 74, North Holland Publishing Co..

[37] Salton, G., Yang, C.S. and Yu, C.T. (1975), "A theory of term importance in automatic text analysis", JASIS 26, 1, 1975.

[38] Weiss, S.F. (1970), "A template approach to natural language analysis for information retrieval", Ph.D. thesis, Cornell University, Department of Computer Science, 1970.

[39] Williamson, R.E. (1974), "Real time document retrieval", Ph.D. thesis, Cornell University, Department of Computer Science, June 1974.

[40] Wong, E. and Chiang, T.C. (1971), "Canonical structure in attribute based file organization", CACM 14, 9, September 1971, pp 593-597.

[41] Yu, C.T. (1973), "Theory of indexing and classification", Ph.D. thesis, Cornell University, Department of Computer Science, August 1973.

[42] Zipf, G.K. (1949), Human Behavior and the Principle of Least Effort, Reading, Mass., Addison-Wesley Publishing, 1949.