A FILE STRUCTURE FOR LARGE DATA BASES

By

Tung Ching Chiang

Memorandum No. ERL-M357

1 September 1972

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Preparation of this Thesis is sponsored by the Army Research Office--
Durham, Grant (AROD) DAHC04-67-C-0046.

TO MY PARENTS

ABSTRACT

This thesis introduces a new file structure which is based on the concept of atoms in a Boolean algebra. Starting with the assumption that the allowable queries are Boolean combinations of keywords, the process of answering a query is treated as evaluating a homomorphism which maps a Boolean algebra of queries into a Boolean algebra of the file. For implementation purposes, a file structure is represented by a nested sequence of partitions of the file, and trees and binary trees become natural choices for the data structure. With this representation, the process of evaluating a query is viewed as a sequence of successive approximations of a Boolean function.

A program incorporating these ideas has been written and run on some randomly generated data; the feasibility of this new file structure is confirmed. Comparison against inverted file structure, with respect to both the storage requirement and the retrieval time, is given.

ACKNOWLEDGEMENTS

The author is deeply indebted to Professer E. Wong for his encouragement and guidance during the research and writting of this dissertation. His advice, at the same time, has been timely, competent and motivating. The author also wishes to thank Professor M. Stonebraker and Professor W. Cooper for their constructive reading of this dissertation.

I wish to thank my friends Mark Chang for his nonstop typing of this thesis and Mary Chen for providing us with a pleasant atmosphere during the final stage of this work.

TABLE OF CONTENTS

I. INTRODUCTION

In recent years, it has become clear that the ever increasing demand for systems which can handle large amounts of data with complex information structures is taxing the resources of existing techniques of data management. Thus, the problem of data base design, more specifically the problem of file organization, has become a problem of considerable importance and urgency in data processing.

Roughly stated, the primary goal of organizing a data file is to enable it to be interrogated via its content. Short of exhaustive search, there are really only two general approaches to achieving this goal. First, where things are stored can be make to depend on what is stored in such a way that access can be facilitated via computation. This approach is exemplified by the familiar hash addressing techniques. Secondly, information in addition to the main data file can be stored and utilized in accessing. Indexing is a familiar example of this approach.

Naturally, how a file is organized should depend on how it is to be used. A fairly general and widely applicable mode of accessing is through keywords

and Boolean combinations of keywords, a keyword being a pair (attribute name, attribute values). Focusing our attention on files which are to be interrogated via keywords and Boolean combinations of keywords, we shall propose a file structure which corresponds closely to a canonical representation of a finite Boolean algebra. We shall consider in detail some ways in which such a structure can be implemented. Comparisions with the well known inverted file and multilist file organizations will be made.

II. EXISTING FILE STRUCTURES

In the field of data management, many articles have been written on the various aspects of file organizations. Some of the specific topics are: studies of existing commercial data management systems [9,12,17,18], classification of the existing file organizations [9,10,11], developments of formal systems or models for file organizations [2,3,4,5,6,20] and discussions on particular file organization techniques [15,21,22]. Different authors use different techniques to decribe files and file organizations. The differences are due either to the differences in their basic conceptual framework or to the authors' preferences in using different terms to describe the same concepts. Codd [4,5] proposed a relational model in which a file is described by a collection of time variant relations and the data base sublanguage is in a form of the first order predicate calculus; McGee [2] used the so-called property classification method to describe a file which can be viewed as a collection of data elements, each element being a (property name, property values) pair; relationships between these collections of data elements are explicitly indicated by the so called record types

and ranks. Hsiao [6] define a record as a subset of the cartisian product A x V, where A is a set of "attributes", V is a set of "values" and developed a formal system to describe file organizations, for example, indexed sequential file and inverted file, and finally defined some retrieval functions to describe procedures for retrieving information from files. Designers of different existing commercial information systems such as GIS, IDS, TDMS, used their own languages to describe the systems they had implemented [9,17,18] . Terminologies that have been used to descrive logical file structures include: hierarchical files which are in forms of trees, heterogeneous files which are in forms of networks; the terminologies used to describe a file include: data element which is a synonym for field, item, element, attribute and property; group which is a synonym for segment, subfile, group of elements [9] . While terminology does not appear to be standard, there is a good deal of agreement on the importance of some of the underlying concepts and on the need for a machine independent and data independent model for files and file structures.

In this Chapter, we make no attempt to standardize the terminology of file organizations. We shall adopt names which to us appear to be most consistent and

descriptive, and in each instance we shall attempt to give a precise definition for the term that we introduce. Finally, we will give a brief survey of some existing file organization techniques.

II.1. DEFINITIONS

Loosely speaking, a file is a collection of records, each record being a collection of properties pertaining to the same individual, item, or object. In its simplest representation, a record can be viewed as an n-tuple of pair (attribute name, attribute value). For example, a record in a personel file may look like the following:

{ (Employee name, J. Smith), (Date of birth, 6/7/40), (Date employed, 7/1/63), ... }

Thus, an attribute can be viewed as a function f_i mapping a subset of the file into a value set, and a file can be viewed as a collection of functions with overlapping domain. For example, in a personel file, "Employee name" may be a name of function mapping from a subset of personel records to a value set which is a set of names, say, { A. Jones, B. John, J. Smith, ... } ; "Date of birth", "Date of employed" may be other names of functions mapping from a subset of personel records to a value set consisting of dates. A

collection of records will be called homogeneous if the same set of attributes are defined on every record in the collection, and will be called inhomogeneous otherwise. For an example of inhomogeneous file, consider a personel file in a university. Suppose there are three subfiles: an employee file which is a collection of functions with names { "Employee name", "Date of birth", "Date of employed" }; a department file which is a collection of functions with names { "Department name", "Population", "Building name", } and a employment file which is a collection of functions { "Department name", "Employee name" }. Note the inhomogeneity in the file, for not all these functions have the same domain.

Clearly, a homogeneous file has the advantage that it admits a tabular representation, e.g.

Employee name	Date of birth	Date of employed
J. Smith	7/1/40	6/2/68
A. Jones	5/2/35	7/15/70
M. Chang	7/21/46	6/16/73
...	...	...
...	...	...

A homogeneous subfile has the further advantage that it can be viewed as a subset of the product space consisting of the product of the value spaces of the attributes involved,i.e., a relation. Because any file can be represented as a set of homogeneous subfiles a file can be viewed as a collection relations, and this is a data base model that is enjoying considerable current popularity [4,8] .

Let R be a collection of records and let f be an attribute mapping R into some value space V. A frequently occuring query has the form: Given the name of the attribute f and a subset S of V, find all records x in R such that $f(x) \in S$. Thus, such a query is specified by a pair (attribute name, a set of attribute values). We shall call such a pair a keyword. For example, (Employee age, less than 35) is a keyword, and so is (Employee number, 12345). Frequently, an attribute f which is a one-to-one map from R into V is designated as the identifying attribute for R, so that f(x) can take the place of x for the reference purposes. Even when there is no such attribute to begin with, it is often convenient to introduce one. A keyword involving the name of an identifying attribute and a single attribute value will be called a primary key. Thus, for example,

(Employee number, 12345) is a primary key if there is a single record in R for each employee number.

The rest of this chapter will be devoted to a survey of some commonly used file organization techniques which are designated to facilitate access via keywords. The main objective of this thesis, to be developed in later chapters, however, will be to present and analyze a class of file structures aimed at retrieval via Boolean combination of keywords.

II.2. EXISTING FILE ORGANIZATION TECHNIQUES

We shall examine the following commonly used file organization techniques: sequential file, indexed sequential file, address calculating, inverted file and multilist file. They are divided into two categories, namely, retrieval via the primary key and retrieval via nonprimary keys, according to the classes of queries that they response to most efficiently. For example, if a file is organized mainly for the retrieval via the primary key, to answer a query involving only nonprimary keys could mean a exhaustive search of the whole file. In practice, combinations of these techniques can be employed in a single file organization [10,11]. For example, in an inverted file, the collection of inverted lists can be viewed as records,

the keyword corresponding to a list can be viewed as the primary key of the record. Thus, the techniques for retrieval via the primary key can be employed to access these inverted lists. In fact, many existing information systems [10,17,18] are using a mixture of these file organization techniques.

II.2.1. RETRIEVAL VIA THE PRIMARY KEY

A. Sequential Method and Indexed Sequential Method:

In the sequential file organization, a file is stored sequentially by the ordering of values of the primary key. When records are to be retrieved, the file is searched sequentially for the specified values of primary key. If the file is on a random access storage device nonsequential techniques, e.g., binary search, can be employed.

In the indexed sequential file organization, records are usually stored randomly in random access storage devices, for example, disks. In addition, a directory consisting of pairs of the form: value of the primary key versus address of the record, is also stored. Usually, the directory entries are ordered according to the ordering of the values of the primary key. Those value-address pairs are referred to as indexes,

which can be grouped into blocks. The maximum value of the primary key within one block can be served as the identifier for the block. Recursively, pairs of the identifier of a block versus the addrss of the block can be considered as indexes. Therefore, a directory may have a hierarchy structure which speeds up searches if the directory is stored on a disk-type storage. Conceptually, the file is partitioned into partitions by the indexes in the directory, each block representing some records whose values of the primary key are in the block. Graphically, the indexed sequential file organization can be illustrated in an example as shown in Fig. 2.1, where numbers represnt values of the primary key, T_{ij} represents the leading address of a block, a_k's represent addresses of records. Each level in the hierarchy represents a partition of the file. Each block in a level represents one member in the partition. For example, at level 2, the partition consists of members: T_{21}, T_{22}, T_{23}, T_{24}. In T_{21}, there are four indexes, number 1,3,7,10 being the values of the primary key of four records whose addresses are a_1, a_2, a_3, a_4 respectively. A query is answered by traversing the directory from top to bottom; at each level values of the primary key specified in the query is compared to the values in

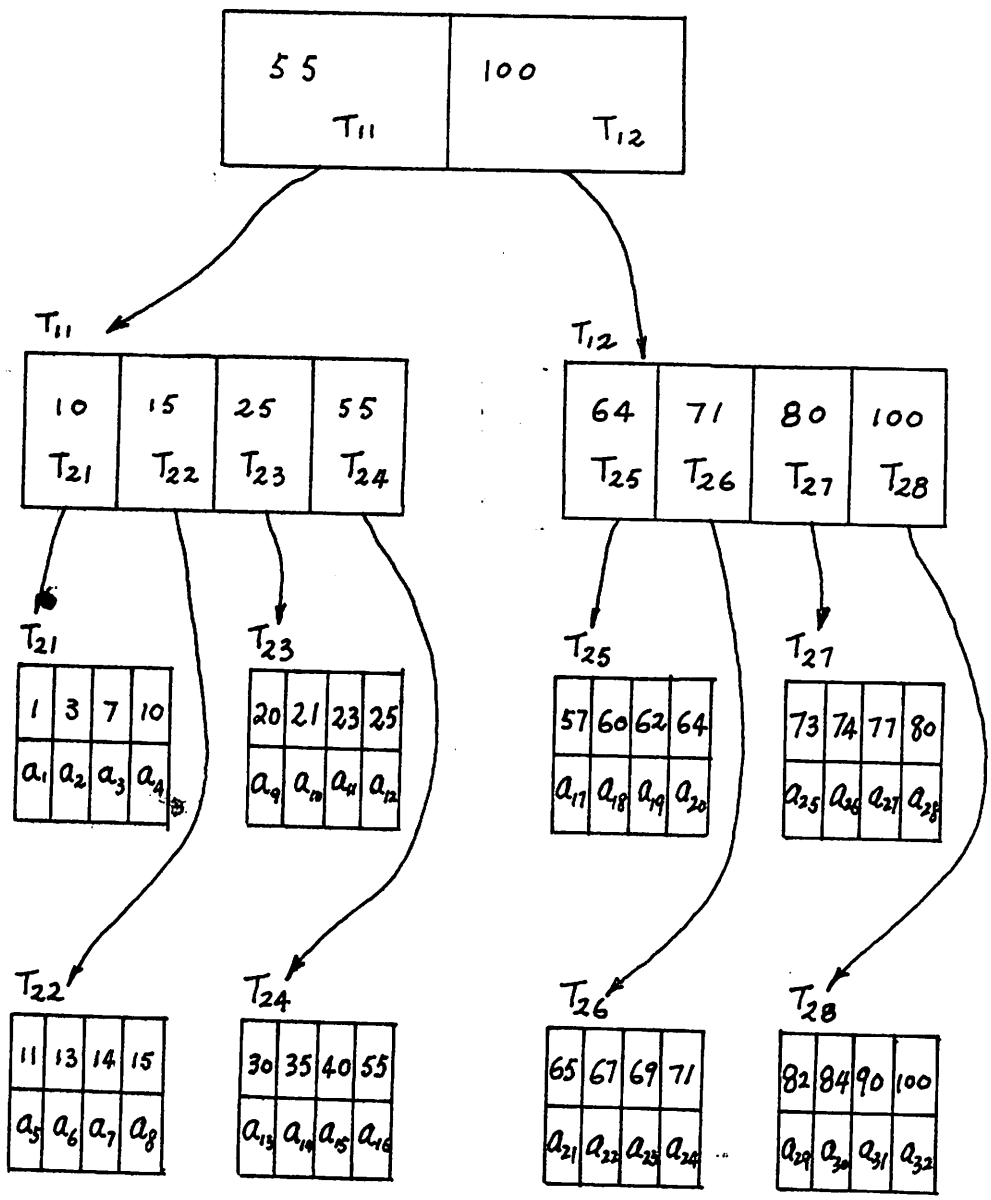

55
100
T11
T12
T11
10
15
25
55
T21
T22
T23
T24
T12
64
71
80
100
T25
T26
T27
T28
T21
1 3 7 10
a1 a2 a3 a4
T23
20 21 23 25
a9 a10 a11 a12
T25
57 60 62 64
a17 a18 a19 a20
T27
73 74 77 80
a25 a26 a27 a28
T22
11 13 14 15
a5 a6 a7 a8
T24
30 35 40 55
a13 a14 a15 a16
T26
65 67 69 71
a21 a22 a23 a24
T28
82 84 90 100
a29 a30 a31 a32

Fig. 2.1

a block. If the values specified in the query are found to be within ranges of some blocks, then the pointers in those blocks are traced and the values in the next level are again compared. When the last level is reached, records whose values of the primary key are matched with those specified in the query are retrieved by tracing the pointers.

Generally, the response time of an indexed sequential file is less than that of the corresponding sequential file. But additional storage space is required for the directory in a indexed sequential file.

B. Address Calculation Method(or Hash Coding)

Let $P = \{1,2,\ldots,p\}$ be the range of the primary key, $M = \{1,2,\ldots, m\}$ be the set of addresses in the storage device, r be the number of records in a file, and assume $p \geq m \geq r$. Let f be a function mapping from P to M. By address calculation, we mean that for a given value p_i in P, apply f to p_i such that $m_i = f(p_i)$ and m_i in M. Usually, f is not an one-one function, i.e.,two different values in P, may be assigned to the same address by f. In this case, the second record may be stored in another location which is chained to the first one by a pointer in a link field in the first record , or by another method.

The advantage of this method is that in most cases, the computation time for f is smaller than the search time for either sequential or indexed sequential method. The disadvantage is that if conflicts of addresses occur very frequently, the price paid for maintaining chains among records will be high. Thus, it is important to choose a mapping f which keeps the conflicts a a minimum. Such a function is difficult to find. It is commonly done by performing one or more arithmatic operations on all or part of the BCD coded representation of the key value and extracting part of resulting code to yield the address. [26] The best function f is then selected after several trials.

II.2.2. RETRIEVAL VIA NONPRIMARY KEYWORDS

Techniques of file organization described in the previous section are incapable of handling queries which contain keywords other than the primary key, without examining every record in the file. In this section we will describe two file organization techniques that can handle queries involved nonprimary keywords without resulting to exhaustive searches.

A. Inverted File

Let $K = \{k_1, k_2, \ldots, k_n\}$ be the keyword set, $L = \{ L(k_i)\ i=1,2,\ldots,n \}$ be the set of keyword lists, each $L(k_i)$ being an ordered list of addresses of all records that have k_i as one of their keywords. Define $D = \{ (k_i; L(k_i))\ \ i=1,2,\ldots,n \}$. An inverted file is the date file together with D, which is referred to as the directory of the inverted file. The term "inverted" can be thought of as inverting a function. Recall that k_i is a (attribute name, attribute value) pair and an attribute can be considered as a function f_i. Let $k_i = (f_i, m_i)$, where $m_i = f_i(x)$ for some x in R, then $L(k_i) = f^{-1}(m_i) =$ (address of $x \mid f(x) = m_i$). Addresses in $L(k_i)$ are referred to as pointers.

Graphically, an inverted file can be illustrated in an example as shown in Fig.2.2, where the symbol $r_i \longrightarrow$ is representing a pointer which is the address of the record r_i.

The allowable queries for the inverted file organization fall into two categories: (1) a single keyword being specified (2) Boolean combinations of keywords. The first type of query can be easily answered by retrieving a list $L(k_i)$ in D, then by tracing the pointers in $L(k_i)$ to fetch the records. As for the second type of queries, Boolean operations

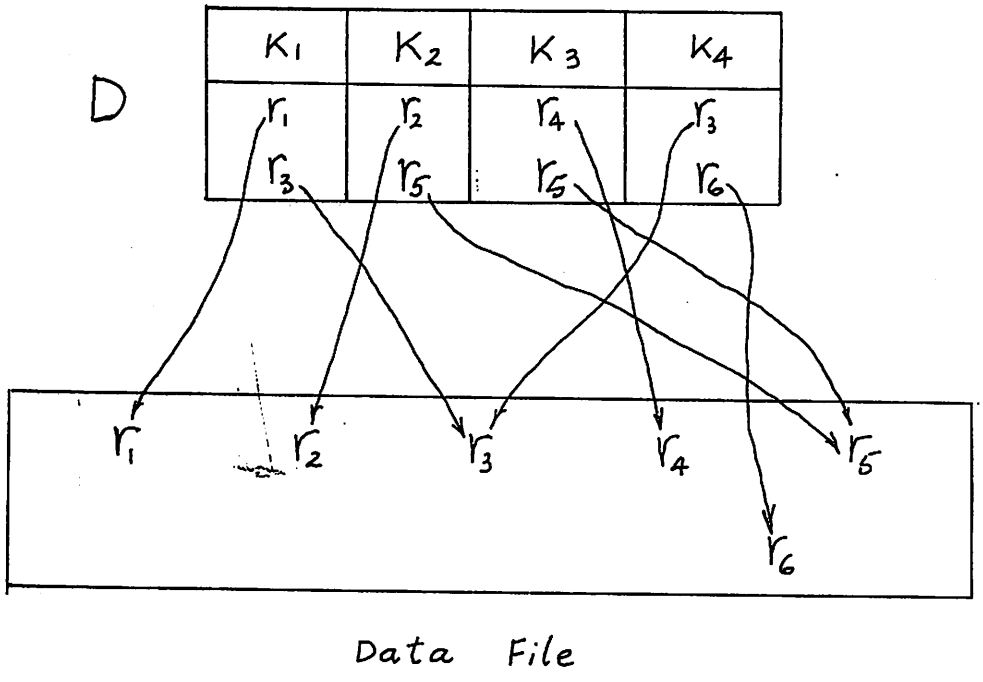


Fig. 2.2

on lists $L(k_i)$ are necessary for queries to be answered.

Example 1: Given a query "$q = k_1$" which is interpreted as "Get all records having k_1". Assuming we have a file as shown in Fig.2.2. The answer to q, denoted by L(q), is $L(q) = L(k_1) = (r_1, r_3)$.

Example 2: Given a query "$q = k_1 \wedge k_4$" which is interpreted as "Get all records having k_1 and k_4". The answer to this query q, denoted by L(q), is $L(q) = L(k_1 \wedge k_4) = L(k_1) \cap L(k_4) = (r_3)$.

Update of an inverted file consists of updating the data file and putting new entries in the directory.

B. Multilist File

Conceptually, the multilist file is the same as the inverted file, in the sense that lists $L(k_i)$'s have to be created. The only difference between these two file organizations is that the ways of implementing those $L(k_i)$'s are different. In the inverted file, the entire list $L(k_i)$ is stored in the directory. In the case of multilist file, only the head of each list is stored in the directory and the rest of the list is threaded through the data file. Records that have the same keyword are chained together by pointers in records.

Let h be an operation on a list $L(k_i)$ such that $hL(k_i)$ = (the first element in $L(k_i)$). The directory of the multilist file is a collection of $(k_i; l_i;$

$hL(k_i)$; i=1,2,...,n), l_i being the length of $L(k_i)$. Records in the data file are modified and represented as $\{ (k_1/p_{j1}, k_2/p_{j2}, \ldots, k_n/p_{jn}), j = 1,2,\ldots,r \}$, where p_{ji} is a pointer in record j, associated with keyword k_i. p_{ji} points to a record which also has k_i. The end of a list can be signified by letting p_{ji} = blank.

Graphically, the multilist file can be illustrated in an example as shown in Fig.2.3, where r_i represents a pointer to r_i. List $L(k_i)$ can be constructed by tracing pointers. For example, $L(k_i) = (r_1, r_2)$, and it is terminated at r_2.

The allowable set of queries for multilist file is the same as that of inverted file, but the procedure for answering a query for the multilist file is somwhat different from that for the inverted file. The first type of query is answered by retrieving the head of a list in the directory and then tracing pointers to get the rest of the list. For the second type of query, records of the shortest list that involves in a query is retrieved, then examine each record for qualification of the query.

Update for multilist file involves putting a record at the end of a list, deleting a record from a list and changing the number representing the length

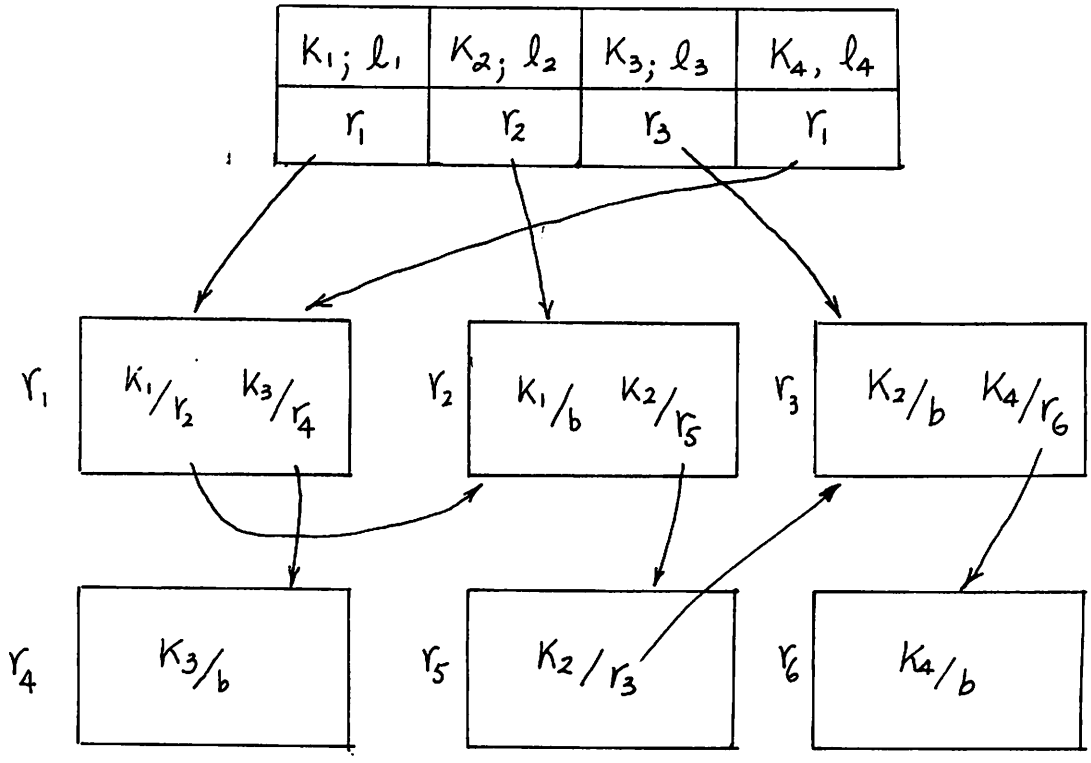

K1; l1
K2; l2
K3; l3
K4, l4
r1
r2
r3
r1
r1
K1/r2
K3/r4
r2
K1/b
K2/r5
r3
K2/b
K4/r6
r4
K3/b
r5
K2/r3
r6
K4/b

Fig.2.3

of a list in the directory.

There is a whole spectrum of file structures that can be constructed between inverted file and multilist file. [6, 11] The length of a list that threaded through the data file can be considered as a design parameter and fixed to a predetermined value to meet some specific requirements of a particular file. These file structures are referred to as controlled list files. Inverted files and multilist files are two extreme cases, the former with list length being one, the latter being infinite.

The basic concepts of file structures described above are very much the same, i.e. lists of keywords are created. Answering a query always involves list processing or examining a large portion of the file. In the next chapters, we will introduce some new concepts of file structure and propose an alternative which is advantageous for large data base with complex Boolean queries.

III. A NEW FILE STRUCTURE

In a file organization that admits all Boolean combination of keywords as its queries, the query language can be represented in a form of propositional calculus, and sets of records are usually retrieved by a sequence of set operations involving union intersection and complementation. We shall show that both the query set and the retrievable set of records are Boolean algebras. And the retrieval procedure can be viewed as the evaluation of a Boolean algebra homomorphism. Concepts of Boolean algebra will be applied to file organization and used as a tool to construct a new file structure. In Section III.1, we shall introduce some definitions and notations related to Boolean algebra and propositional calculus. Theorems of Boolean algebra will be stated without proof. Postulates of Boolean algebra have been given by many other authers and we shall not repeat them here. [33,34]

III.1. BOOLEAN ALGEBRA AND PROPOSITIONAL CALCULUS

Terms and notations in connection with Boolean algebra and propositional calculus are shown as follows:

Notations	Boolean algebra	Prop. calculus
$\vee$	join	or
$\wedge$	meet	and
$-$	complementation	not
$\Leftrightarrow$	biconditional	biconditional
$\Rightarrow$	implication	implication
0	null element	false sentence (or fornula)
1	universal element	valid sentence (or formula)

$\Rightarrow$ and $\Leftrightarrow$ are defined as follows:

For any α, β in a Boolean algebra or in a set of well-formed formula in a propositional calculus

$$\alpha \Rightarrow \beta = (\overline{\alpha} \vee \beta)$$
$$\alpha \Leftrightarrow \beta = (\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$$

Notations $\cup, \cap, -$ are used as union, intersection, and complementation operations respectively in set theory.

Definitions in connection with Boolean algebras are given as follows:

Subalgebra: Let A be a Boolean algebra. A subset $B \subseteq A$ is called a subalgebra of A if B is closed under $\wedge, \vee, -$ and 0,1 are in B.

Generators: Let E be an arbitrary subset of a Boolean algebra A. The intersection of all those subalgebras

that contain E is a subalgebra. That intersection, sayB, is the smallest subalgebra of A that contains E. The subalgebra B is called the subalgebra generated by E. A generating set E is called a set of generators of B.

Homomorphism: A Boolean Homomorphism is a mapping f from a Boolean algebra B, to a Boolean A, such that

$$(1)\ f(p \wedge q) = f(p) \wedge f(q)$$

$$(2)\ f(p \vee q) = f(p) \vee f(q)$$

$$(3)\ f(\overline{p}) = \overline{f(p)}$$

for all p and q in B. The kernel of a homomorphism f is defined as $(x|f(x)=0)$.

Free Algebras: Let E be a subset in a Boolean algebra B. If the elements in E satisfy no condition except those that can be derived from the set of postulates, it is a natural way to say E is free. A Boolean algebra is called free if it has a free set of generators. Let A, B, be two free Boolean algebras, E_1, E_2 be the sets of generators respectively. If there is a function f mapping from E_1 to E_2, then there exist an unique homomorphism H mapping from A to B, such that $H(j) = f(j)$ for all j in E_1 .

Order: It is easy to show that $p \wedge q = q$ iff $p \vee q = p$. we write $p \geq q$ or $q \leq p$ if $p \wedge q = q$ or equivalently

$p \vee q = p$. $\leqslant$ is a partial ordering relation.

Atoms: An atom of a Boolean algebra B is an element that has no nontrivial subelements defined by the partial order $\leqslant$. Let $q \in B$, q is an atom of B if $q \neq 0$ and if $p \leqslant q$, then $p = 0$ or q. A Boolean algebra which has atoms is referred to as atomic Boolean algebra.

It can be shown that a finitely generated Boolean algebra is finite, and in fact, the number of elements in a Boolean algebra with n generators is $\leqslant 2^{2^n}$. Every finitely generated Boolean algebra is atomic and the number of atoms of a Boolean algebra with n generators is $\leqslant 2^n$. Furthermore, if the Boolean algebra is freely generated by n generators, then the number of atoms is exactly equal to 2^n.

Let $E = (e_1, e_2, \ldots, e_n)$ be a set of generators of B. An atom c has the form $c = \bigwedge_{i=1}^{n} e_i^*$; $e_i^* = e_i$ or $\bar{e}_i$. Let $C = (c_1, c_2, \ldots, c_s)$ be the set of all atoms of B and $I = (1, 2, \ldots, s)$ be the set of subscripts of c_i. Every element p in B can be represented in the form

$$p = \bigvee_{i \in J} c_i = (\text{Joins of some atoms in C})$$

where J is a subset of I.

Propositional calculus can be viewed as a formal system $\mathcal{A} = (\Phi, \Sigma, -, \vee, \wedge)$, Σ is the set of statement letters, Φ is the set of all well-formed formulas,

such that $\Sigma \subseteq \Phi$ and Φ is closed under $-, \wedge, \vee$ [35]
Let $\mathcal{S} = (W, W, \bar{}, \vee, \wedge)$ be a special formal system, with $W = (0, 1)$. Define

X	$\bar{X}$
0	1
1	0

$\wedge$	0	1
0	0	0
1	0	1

$\vee$	0	1
0	0	1
1	1	1

If there is a function $j: \Sigma \longrightarrow W$, then j can be extended to a homomorphism $v_j : \Phi \longrightarrow W$.

Definitions: j is said to satisfy $b \in \Phi$ if $v_j(b) = 1$. b is said to be logically valid if every j satisfies b. Two formulas ϕ and b are said to be logically equivalent if $v_j(\phi) = v_j(b)$ for all j, we write $\phi \equiv b$. Note that $\equiv$ is an equivalence relation.

Let $\Sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$ be a set of statement letters in a given order. A truth function or a Boolean function b* associated with a formula b in Φ, mapping from $W^n \longrightarrow W$, $W^n = WxWx...xW$, is defined as follows:

For any $(p_1, p_2, \ldots, p_n)$ in W^n, $b^*(p_1, p_2, \ldots, p_n) = v_j(b)$, where j is a mapping from Σ to W, for which $j(\sigma_1) = p_1$, $j(\sigma_2) = p_2$, ..., $j(\sigma_n) = p_n$. We call b* the n-place truth function or the n-place Boolean function defined by b in Φ. A truth table of a truth function b* is the tabular representation of the truth function b*. The truth table of b*

consists of two sets of n-tuples: $s_1 = ((p_1, p_2, \ldots, p_n)\ b^*(p_1, p_2, \ldots, p_n)=1)$, $s_2 = ((p_1, p_2, \ldots, p_n)\ b^*(p_1, p_2, \ldots, p_n)=0)$. s_1, s_2 are referred to as the on-set and the off-set of b* respectively.

It can be shown that the set of equivalence classes of formulas is a Boolean algebra, or equivalently, the set of all Boolean functions defined by all formulas in is a Boolean algebra.

III.2. ATOMS OF A FILE:

Let $K= (k_1, k_2, \ldots, k_n)$ be the set of keywords of a file F. Let the query set Q be the set of all Boolean combinations of keywords, i.e., every q in Q is made up of elements in K , connected by $\bar{}, \wedge, \vee$ in a obvious way. The formal system $\langle Q, K, \bar{}, \vee, \wedge \rangle$ is a propoitional calculus. After identification of equivalent queries, Q is the free Boolean algebra B(K), generated by the set K. Let L be a function mapping from K to the collection of all subsets of F, such that $L(k_i)$ is the set of all records that have k_i as a keyword. Let $B(\mathcal{L})$ denote the Boolean algrbra generated by $\mathcal{L} = (L(k_i); i=1,2,\ldots,n)$. L can be considered as a function mapping from K to $B(\mathcal{L})$. L can be uniquely extended to a homorphism mapping from B(K) to $B(\mathcal{L})$. We shall use the same notation L to denote this

homomorphism. Once $L(k_i)$ are defined , every q in Q can be answered by finding the homomorphism image of q in $B(\mathcal{L})$. In other words, for any q in Q, L(q) is the set of records that satisfy q. For example, let $q = k_1 \wedge k_2$, then $L(q) = L(k_1) \wedge L(k_2)$. Once $L(k_1), L(k_2)$ are defined, L(q) can be obtained by the intersection of two lists $L(k_1)$ and $L(k_2)$. That is why, in the inverted file, by storing lists of addresses corresponding to $L(k_i)$ we can retrieve every set in $B(\mathcal{L})$. Any other collection C of subsets in $B(\mathcal{L})$ can serve the purpose, provided that $B(C) = B(\mathcal{L})$. The question is what lists should be stored. We hope to show that in many situations, the lists representing atoms in $B(\mathcal{L})$ are the best choices. Note that $B(\mathcal{L})$ may not be free, i.e., there may be some implicit relationship in addition to those universal ones, existing between elements in the generating set $\mathcal{L}$. In particular, intersection of some elements in $\mathcal{L}$ may be the null element. This implies that the number of atoms of $B(\mathcal{L})$ may be much less than 2^n. This is a favorable situation, because the lists that we store will correspond to atoms. It is conjectured that other relationships existing among elements of

may be worthwhile to be explored and taken advantage of, but in this present work they are not considered.

The set of atoms of a file with respect to a set of keywords is defined to be the set of atoms of $B(\mathcal{L})$. Let D be the set of atoms of B(K), C be the set of atoms of $B(\mathcal{L})$, then C is the range of the restriction of the homomorphism L on D. Standard results in Boolean algebra [33] imply that the atoms of a file have the following properties:

(1) Atoms of a file are the minimal retrievable set of records in a file, for a given query set.

(2) Atoms of a file are pairwise disjoint sets, i.e. $c_i \cap c_j \neq 0$, for any two distinct atoms c_i, c_j.

(3) Any query in the query set can be answered by using only union operations on atoms of the file.

(4) Atoms of a file are the nonempty sets obtained by $\bigcap_{i=1}^{n} L^*(k_i)$, $L^*(k_i) = L(k_i)$ or $\overline{L(k_i)}$.

We now propose a file structure in which lists of addresses of records which are stored correspond to the atoms of the file. This file structure has the following advantages: [22]

(a) Each address appears on one and only one list. Hence the number of addresses to be stored is always less than the total number of addresses in ($L(k_i)$, i=1,2,...,n).

(b) Every set to be retrieved is a union of disjoint atoms. We never need to take intersection

and we never need to eliminate duplications in taking union.

(c) The computation procedure in translating an arbitrary Boolean formula of keywords into a union of atoms is exceedingly simple (but may be time consuming).

Assertions (a) and (b) are obvious consequences of (2), (3), (4). Assertion (c) is justified by considering the standard procedure for translating a Boolean formula into its <u>developed disjunctive normal form</u>.

For an example, consider a file F with 10 records denoted by $F = (1,2,\ldots,10)$. Let $K = (k_1,k_2,k_3,k_4)$ be the set of keywords. The query set Q is the Boolean algebra B(K) generated by K. For the purpose of this example, we do not distinguish between a record and its address. Let $\mathcal{L} = (L(k_1), L(k_2), L(k_3), L(k_4))$ be the set of lists of records that have k_1,k_2,k_3,k_4, respectively. The collection of all retrievable sets is the Boolean algebra $B(\mathcal{L})$ generated by $\mathcal{L}$. Let F be represented as a tabular form as shown in Table 3.1, where an entry '1' indicates the record belongs to $L(K_i)$ and an entry '0' indicates it does not. It is clear that we have

$$L(k_1) = (1,2,4,6,9)$$
$$L(k_2) = (1,3,4,6,7)$$

	k_1	k_2	k_3	k_4
1	1	1	0	0
2	1	0	1	0
3	0	1	1	0
4	1	1	0	0
5	0	0	1	1
6	1	1	0	0
7	0	1	1	0
8	0	0	1	1
9	1	0	1	0
10	0	0	1	1

Table 3.1.

	$L(k_1)$	$L(k_2)$	$L(k_3)$	$L(k_4)$
c_1	1	1	0	0
c_2	1	0	1	0
c_3	0	1	1	0
c_4	0	0	1	1

Table 3.2.

$L(k_3)$ = (2,3,5,7,8,9,10)

$L(k_4)$ = (5,8,10)

Let C be the set of atoms of the file. The atoms can be obtained easily from examining Table 3.1.

$c_1 = L(k_1) \cap L(k_2) \cap \overline{L(k_3)} \cap \overline{L(k_4)}$ = (1,4,6)

$c_2 = L(k_1) \cap \overline{L(k_2)} \cap L(k_3) \cap \overline{L(k_4)}$ = (2,9)

$c_3 = \overline{L(k_1)} \cap L(k_2) \cap L(k_3) \cap \overline{L(k_4)}$ = (3,7)

$c_4 = \overline{L(k_1)} \cap \overline{L(k_2)} \cap L(k_3) \cap L(k_4)$ = (5,8,10)

$C = (c_1, c_2, c_3, c_4)$, c_i are in an arbitrary order. The set C of all atoms can be represented in a tabular form as shown in Table 3.2., where an entry '1' indicated $L(k_i)$ appears in the disjunctive form, '0' indicated $\overline{L(k_i)}$ appears in the disjunctive form.

Let $q = (k_1 \wedge k_2 \wedge \overline{k}_3)$ be a query in Q. The answer to q is

$$\begin{aligned} L(q) &= L(k_1 \wedge k_2 \wedge \overline{k}_3) \\ &= L(k_1) \cap L(k_2) \cap \overline{L(k_3)} \\ &= (L(k_1) \cap L(k_2) \cap \overline{L(k_3)} \cap L(k_4)) \\ &\quad \cup (L(k_1) \cap L(k_2) \cap \overline{L(k_3)} \cap \overline{L(k_4)}) \\ &= L(k_1) \cap L(k_2) \cap \overline{L(k_3)} \cap \overline{L(k_4)} \\ &= (1,4,6) \end{aligned}$$

or

$$\begin{aligned} L(q) &= L(k_1 \wedge k_2 \wedge \overline{k}_3) \\ &= L((k_1 \wedge k_2 \wedge \overline{k}_3 \wedge \overline{k}_4) \vee (k_1 \wedge k_2 \wedge \overline{k}_3 \wedge \overline{k}_4)) \\ &= (L(k_1) \cap L(k_2) \cap \overline{L(k_3)} \cap \overline{L(k_4)}) \\ &\quad \cup (L(k_1) \cap L(k_2) \cap \overline{L(k_3)} \cap L(k_4)) \end{aligned}$$

$$= L(k_1) \cap L(k_2) \cap \overline{L(k_3)} \cap \overline{L(k_4)}$$
$$= (1,4,6)$$

From the example above, we observe that to answer a query q we can apply the homomorphism L to q and expand the resulting formula into its developed disjunctive normal form; the nonempty clauses will be the atoms that are dominated by L(q). Alternatively, we can expand q into its disjunctive normal form and then apply L to it. Though the procedure for expanding a formula into its disjunctive normal form is straightforward, the procedure for determining which clauses represent atoms may be time consuming. A simple procedure for doing this is to intersect the list of clauses in the disjuntive normal form of q with the list of atoms of the file; those ones that match are atoms dominated by L(q). Obviously, if the number of empty clauses in a normal form is large (very often it is), this procedure will be very inefficient, in the sense that the whole set of atoms has to be compared against a long list of clauses and only a small portion of the set of atoms contained useful information pertinent to the query. In general, a procedure for eliminating clauses corresponding to the null element in $B(\mathcal{L})$ for a query q can be viewed as a procedure for finding atoms of the meet of two

Boolean formulas in B(K), or equivalently, for finding the meet of two Boolean functions.

There are many formulas b in B(K) such that L(b) = C, the set of all atoms. Let f be such a formula with the further property that every clause in the developed normal form of f corresponds to an atom in $B(\mathcal{L})$, i.e., $L(\varphi) \neq o$, for all clauses φ in the developed disjunctive normal form of f. Therefore there is an one-one relationship between clauses of f and atoms of $B(\mathcal{L})$, we can represent the set of atoms C of $B(\mathcal{L})$ by f. Such a formula can be obtained easily. In fact, let the tabular representation of C be the on-set of f^*, i.e., $(p_1,p_2,\ldots,p_n \; f^*(p_1, p_2,\ldots,p_n) = 1)$. Let f^* denote the truth function associated with f. Then, f can be constructed from the on- set of f^*. For example, consider the tabular representation of C as shown in Table 3.2.

$$f = (k_1 \wedge k_2 \wedge \overline{k}_3 \wedge \overline{k}_4) \vee (k_1 \wedge \overline{k}_2 \wedge k_3 \wedge \overline{k}_4)$$
$$\vee (\overline{k}_1 \wedge k_2 \wedge k_3 \wedge \overline{k}_4) \vee (\overline{k}_1 \wedge \overline{k}_2 \wedge k_3 \wedge k_4)$$

is a Boolean formula associated with a truth function f^* that has a on-set $(f^{*-1}(1) = (1100,1010,0110,0011))$.

The clauses in q corresponding to atoms are those clauses in $f \wedge q$. Therefore to find all nonempty clauses in q is to find all clauses in $f \wedge q$. It can be shown that all clauses in the developed disjunctive

normal form of $f \wedge q$ are not in the kernel of the homomorphism L. For if there is a clause φ in $f \wedge q$ such that $L(\varphi) = 0$, and since φ must be a clause in f, then an element of L(f) is equal to 0, a contradiction. One way to find all clauses in $f \wedge q$ is to expand both f and q into their developed disjunctive normal form individually, then the meet of the two normal forms will consists of all the clauses in $f \wedge q$. This is how it is done in the simple procedure stated above . If the number of clauses in q is large, execution time of this simple procedure will be large and require a large block of working space.

There are many other procedures which will serve the purpose for finding clauses in $f \wedge q$. An alternative that involves decomposition of Boolean function, or equivalently, partitioning of a file will be given in the next section. We hope to show that it will often be more efficient than the simple procedure.

III.3. PARTITIONS OF A FILE:

In this section, we will present a method for partitioning a file into a sequence of partitions and a procedure for finding clauses in $f \wedge q$. The procedure will consist of a sequence of steps, at

each step only a portion of a partition of the file has to be searched and a Boolean function of only m variables ($m < n$) has to be dealt with.

Let $K = (k_1, k_2, \ldots k_n)$ be the set of keywords, $\mathcal{L} = (L(k_1), L(k_2), \ldots L(k_n))$, $B(\mathcal{L})$ be the algebra generated by $\mathcal{L}$. We are to find a sequence of sets of generators $G_0, G_1, G_2, \ldots, G_m = \mathcal{L}$, $G_i \subseteq B(\mathcal{L})$, for $i=1,2,\ldots,m$, such that $B(G_0) \subseteq B(G_1) \subseteq B(G_2) \ldots \subseteq B(G_m) = B(\mathcal{L})$. The atom-set C_i of each $B(G_i)$ is a partition of the file. Therefore, corresponding to the sequence of Boolean algebras we have a sequence of partitions $P_0, P_1, P_2, \ldots, P_m$ on the file; P_{i+1} is a refinement of P_i. The procedure for answering a query q is presented as a flow chart as shown in Fig.3.1. At each step i, instead of the whole file, only a subset D_i of C_i has to be examined to get a subset D_{i+1} of C_{i+1} such that $L(q) \subseteq \bigcup_{x \in D_{i+1}} x$. Graphically, assuming m = 2, we can show it in Fig.3.2. Let us represent sets of atoms in a two dimensional surface within the largest square, there are squares with three different sizes, labelled as 1 , 2 , 3 respectively. There are nine 1 squares in the largest square, nine 2 squares in each 1 square, nine 3 squares in each 2 square. For the sake of simplicity, we only draw a portion of all squares.

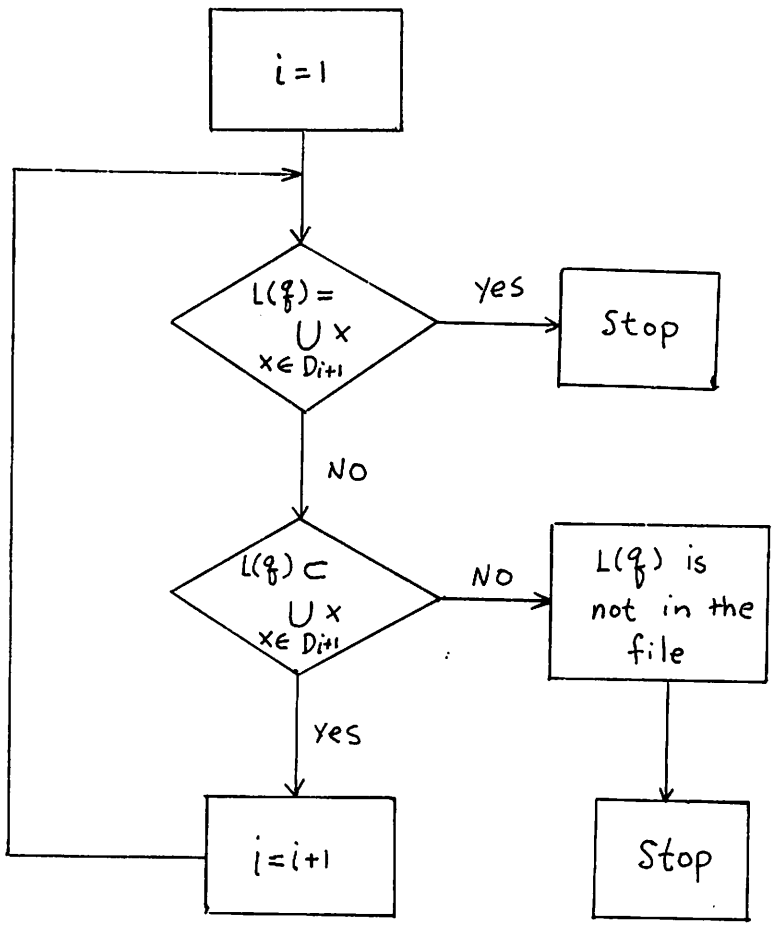

i=1
L(q) = ∪ x, x∈ Di+1
yes
Stop
NO
L(q) ⊂ ∪ x, x∈ Di+1
NO
L(q) is not in the file
yes
i=i+1
Stop

Fig. 3.1

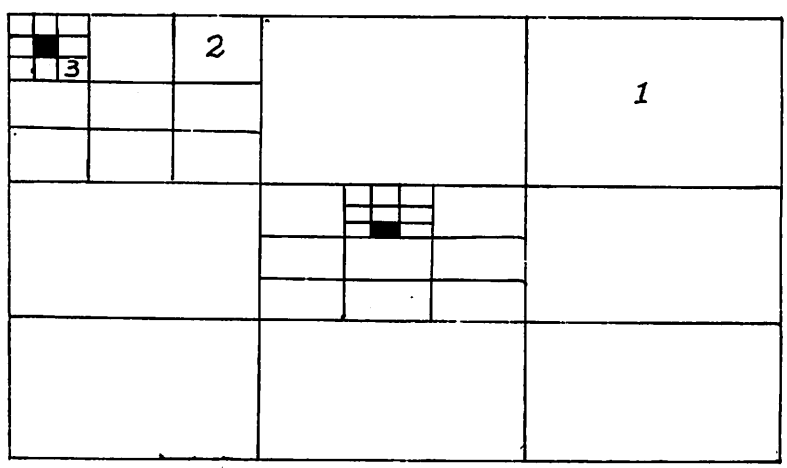

3
2
1

Fig. 3.2

Let the set of all 1 squares represent the set of atoms of $B(G_0)$, the set of all 2 squares represent the set of atoms of $B(G_1)$, the set of 3 squares represent the set of atoms of $B(G_2)$. Squares encircled with heavy lines are those satisfying $L(q) \subseteq \bigcup_{x \in D_{i+1}} x$, finally, squares that are blackened are atoms dominated by $L(q)$.

Answering a query in this way can also be viewed as successively approximating a Boolean function. Let $f_0, f_1, f_2, \ldots, f_m = f$, be the Boolean functions defined by the sets of atoms $C_0, C_1, C_2, \ldots, C_m = C$ respectively. The tabular representation of C_i is the on-set of f_i, i.e. $C_i = ((p_1, p_2, \ldots, p_n) / f_i(p_1, p_2, \ldots, p_n) = 1)$. The sequence of $f_0, f_1, \ldots, f_m$ satisfies the relation

$$f_0 \geqslant f_1 \geqslant f_2 \geqslant \ldots \geqslant f_m = f$$

The procedure for finding clauses in $f \wedge q$ can be viewed as the evaluation of $L(\ldots(f_0 \wedge q) \wedge f_1) \wedge f_2) \ldots \wedge f_m)$, each $(\phi) \wedge f_i$ is an approximation of $f \wedge q$, i.e. $(\phi) \wedge f_i \geqslant f \wedge q$.

Now, a method for constructing such a sequence of Boolean algebras is described. First, the notion of partition of the keyword set is introduced. By the partition of the keyword set K we mean an ordered pair of collections denoted by (Y/Z), where $Y = (y_1, y_2, \ldots, y_s)$, $Z = (z_1, z_2, \ldots, z_t)$, $Y \cup Z = K$, y_i, z_j

in K for i=1,2,...,s and j=1,2,...,t with repetitions allowed, i.e. $s+t \geq n$. When there is no repetition in Y and Z, Y/Z is said to be disjunctive, otherwise nondisjunctive.

Let $Y/Z = (y_1, y_2, \ldots, y_s / z_1, z_2, \ldots, z_t)$ be a disjunctive partition of K. Let g_{m-1} be an arbitrary Boolean function mapping W^t into W, W=(0,1) with z_1, $z_2, \ldots, z_t$ as variables. Let $G_{m-1} = (L(y_1), L(y_2), \ldots, L(y_s), L(g_{m-1}))$. It can be shown that $B(G_{m-1}) \subseteq B(\mathcal{L})$, and if we let f_{m-1} denote the Boolean function defined by the set of atoms of $B(G_{m-1})$, then we have $f_{m-1} \geqslant f$. f_{m-1} is said to be an approximation of f and f is said to be approximated by f_{m-1}. By consecutively selecting disjunctive partitions of generating sets, a nested sequence of Boolean algebras $B_0 \subseteq B_1 \subseteq B_2 \subseteq \ldots \subseteq B_m$ can be obtained, where $B_i = B(G_i)$ i=1,...,m and $B(G_m) = B(K)$. Corresponding to sets of atoms $C_0, C_1, \ldots, C_m$ of $B_0, B_1, \ldots, B_m$ respectively, there is a sequence of Boolean functions, $f_0, f_1, \ldots, f_m$ such that

$$f_0 \geqslant f_1 \geqslant f_2 \geqslant \ldots \geqslant f_m = f$$

Example: Consider a file F with a keyword set $K = (k_1, k_2, k_3, k_4)$ and the set of atoms $C = (c_1, c_2, \ldots, c_7)$. Let f be the Boolean function corresponding to C, such that the tabular representation of C is the on-set of f which is shown in Table 3.3.

X	k_1	k_2	k_3	k_4
c_1	1	1	0	0
c_2	0	1	1	0
c_3	1	0	0	0
c_4	0	1	1	1
c_5	0	1	0	1
c_6	1	0	1	1
c_7	1	0	0	1

Table 3.3.

Let g_2 be a Boolean function, with k_3, k_4 as variables, defined by the truth table shown in Table 3.4.

X_1	X_2	g_2
0	0	1
0	1	1
1	0	0
1	1	0

Table 3.4.

Let $Y_1/Z_1 = (k_1, k_2, /k_3, k_4)$ be a disjunctive partition of K, $G_1 = (L(k_1), L(k_2), L(g_2))$. By applying g_2 to the last two columns in Table 3.3, the tabular representation

of the set C_1 of atoms of $B(G_1)$ is obtained in Table 3.5.

	k_1	k_2	g_2	
a_1	1	1	1	(c_1)
a_2	0	1	0	(c_2, c_4)
a_3	1	0	1	(c_3, c_7)
a_4	0	1	1	(c_5)
a_5	1	0	0	(c_6)

Table 3.5.

There are five elements in C_1, namely, $C_1=(a_1,a_2,a_3, a_4,a_5)$. The entries in the last column are atoms of $B(\mathcal{L})$ contained in each atom of $B(G_1)$. Next, let g_i be a Boolean function with k_1,k_2 as variables defined by the truth table shown in Table 3.6.

X_1	X_2	g_1
0	0	1
0	1	1
1	0	0
1	1	0

Table 3.6

Let $G_0 = (L(g_1), L(g_2))$. By applying g_1 to the first

two columns in Table 3.5, the tabular representation of the set C_0 of atoms of $B(G_0)$ is obtained and shown in Table 3.7.

	g_1	g_2		
b_1	0	1	(c_1, c_3, c_7)	(a_1, a_3)
b_2	1	0	(c_2, c_4)	(a_2)
b_3	1	1	(c_5)	(a_4)
b_4	0	0	(c_6)	(a_5)

Table 3.7

There are four elements in C_0, namely $C_0=(b_1, b_2, b_3, b_4)$. The entries in the last column are atoms of $B(G_1)$ contained in each atom of $B(G_0)$. The entries in the second last column are atoms of $B(\mathcal{L})$ contained in each atom of $B(G_1)$. Let Table 3.5 correspond to the on-set of a Boolean function f_1, Table 3.7 correspond to the on-set of a Boolean function f_0. It is clear that $f_0 \geqslant f_1 \geqslant f$. Since $b_i = \bigcup_{a_j \in c_i \subseteq C_i} a_j$, $\forall\, b_i$ in C_0 and $a_i = \bigcup_{c_j \in C_i \subseteq C} c_j$, $\forall\, a_i$ in C_1, it can be seen that $B(G_0) \subseteq B(G_1) \subseteq B(\mathcal{L})$. Note that C_0 is a coarser partition than C_1, C_1 is a coarser partition than C.

Let us use this example to illustrate a procedure for answering a query as follows. Let $W^* = (0,1,*)$.

Let g_1, g_2 be extended to functions mapping from W^{*2} to W^*, g_1, g_2 are redefined as

g_1	0	1	*
0	o	o	o
1	0	0	0
*	*	*	*

g_2	0	1	*
0	1	1	1
1	0	0	0
*	*	*	*

Since every query can be written as one of its disjunctive normal forms, we consider queries involving the Boolean connector 'meet' only. For each such query q, we create a vector $q_2 = (d_1, d_2, d_3, d_4)$. $d_j=1$, if k_j is presented in q; $d_j=0$, if $\overline{k}_j$ is presented in q; $d_j=*$, if neither k_j nor $\overline{k}_j$ is in q. From q_2 we obtain $q_1 = (d_1, d_2, p_2)$, where $p_2=g_2(d_3,d_4)$. From q_1 we obtain $q_0=(p_1,p_2)$. Before proceeding, we must define an operator D. Let E_1, E_2, E_3 be some vector spaces. D is defined as a map from $E_1 x E_2$ to E_3 such that for any V D_1, U D_2, where $V = (v_1, v_2, \ldots, v_k)$, $U = (u_1, u_2, \ldots, u_k)$, $D(V;U) = (v_1-u_1, v_2-u_2, \ldots, v_k-u_k)$, where "-" is defined by

-	0	1	*
0	0	1	0
1	1	0	0
*	0	0	*

An atom x is said to be relevant to the query q if $x \wedge q \neq 0$. It can be shown that an atom x in C_i is relevant to q if $D(x,q_i) = 0$, where 0 is the zero vector (000..0). Therefore the set of all relevant atoms in C_i to q is the set $X_i = (x \text{ in } C_i / D(x_i,q)=0)$. X_i can be obtained recursively without searching the whole C_i. Define $Y_i = (y \text{ in } C_i / x \supseteq y, x \text{ in } X_{i-1})$, then $X_i = (y \text{ in } Y_i / D(y,q)=0)$, with $X_0 = (x \text{ in } C_0 / D(x,q_0)=0)$.

In our example above, let $q=k_1 \wedge \overline{k_3}$, a vector $q_2=(1 * 0 *)$ is created. By mapping q_2 to the last two components of q_2, we have $q_1=(1 * 1)$. By applying g_1 to the first two components of q_1, we have $q_0=(0\ 1)$. We have $Y_0=C_0$, $X_0=(x \text{ in } C_0/D(x,q_0)=0)=(b_1)$. Then in turn, we can obtain the following sets

$Y_1= (y \text{ in } C_1/x \text{ in } X_0,\ x \supseteq y) = (a_1,a_3)$

$X_1= (y \text{ in } Y_1/D(y,q_1)=0) = (a_1,a_3)$

$Y_2= (y \text{ in } C_2/ x\ y,\ x \text{ in } X_1) = (C_1,C_3,C_7)$

$X_2= (y \text{ in } Y_2/ D(y,q_2)=0) = (C_1,C_3,C_7)$

$L(q) = X_2$.

Let us consider another query $q=k_2 \wedge k_3$. By following the same procedure as before we obtain

$q_2 = (* 1 1 *)$, $q_1 = (* 1 0)$, $q_0 = (* 0)$

$X_0 = (b_1,b_2,b_3,b_4)$

$$Y_1 = (a_1, a_2, a_3, a_4, a_5)$$
$$X_1 = (a_2)$$
$$Y_2 = (c_2, c_4)$$
$$X_2 = (c_2, c_4)$$
$$L(q) = (c_2, c_4) = X_2.$$

III.4. DISCUSSION:

Let us define a measure of efficiency coefficient of this file structure corresponding to a certain query by

$$\eta = \left(\sum_{i=1}^{m-1} \frac{|X_i|}{|Y_i|} \right) \Big/ \left(m-1 \right)$$

where $|X_i|$, $|Y_i|$ are the number of elements in X_i, Y_i respectively. η is the average ratio of the number of relevant atoms versus the number of atom to be searched. For the example in III.3, the coefficient for the first query is $\eta = (1/4+1+1)/3 = 0.75$, for the second query $\eta = (1+1/5+1)/3 = 2.2/3 = 0.73$.

In general, the coefficient is high for a certain class of queries and low for others, depending on how the atoms in each algebra are distributed. For a given file, distribution of atoms in each algebra is solely determined by the choice of sets of generators. If the frequencies of usages of queries are known a priori, then, by a certain choice of sets of generators,

the file structure can be tailored to the one that responds most efficiently to those queries that are most frequently used. The response time to a query is closely related to the coefficient η. It depends on how the file structure is implemented and what kind of hardware is being used. But it also depends on the distribution of atoms. Therefore the choice of sets of generators plays an important role in improving both the efficiency and the response time for the file structure.

An extension of this can be made by considering the nondisjunctive partitions of the keyword set. Redundant information may be created as a result of duplications of keywords in the disjunctive partitions. But it might improve the efficiency of the file structure with respect to some classes of queries. Another extension can be made if we consider the resulting set G_0 of Boolean functions $(g_1, g_2, \ldots, g_{m-1})$ as a set of new keywords and apply the procedure to G_0 as is was applied to K, we will have a second sequence of Boolean algebras whose set of atoms correspond to some coarser partitions of $B(G_0)$. Therefore by repeated application of this procedure, a file can be partition into any number of sequences of partitions. When the file and the keyword set

are extremely large, this may result in some important computational advantages.

IV. IMPLEMENTATION AS A TREE

The file structure involving a nested sequence of partitions, as discussed in the preceeding chapter, leads to the idea of implementing the proposed file structure as a tree structure. Each level in the tree corresponds to a Boolean algebra in the sequence, and the nodes at that level represent the atom of the corresponding Boolean algebra in a one-to-one manner. Succeeding levels in the tree correspond in an obvious and natural way to succeeding Boolean algebras in the nested sequence, and descending nodes represent splitting of atoms in an equally natural way.

First of all, we will review some notions of tree and binary trees, as they will be used in our implementation.

IV.1 TREES AND BINARY TREES

The definitions of trees and binary trees that we introduce here are taken from Knuth [36], in which trees and binary trees are defined recursively as follows:

A tree is defined as a finite set T of one or more nodes such that

(a) there is one special node in a tree called the root of the tree, denoted by root(T);

(b) the remaining nodes (excluding the root) are partitioned into $m \geqslant 0$ disjoint sets $T_1,\ldots,T_m$ and each of these sets in turn is a tree. The trees $T_1,\ldots,T_m$ are called the subtrees of T.

In a tree T, a node n_2 is called a son of another node n_1, if there is a branch connecting from n_1 to n_2, and n_1 is called the father of n_2. Two nodes n_1 and n_2 in a tree T are called brothers if n_1 and n_2 are sons of the same node. The number of sons of a node n is called the degree of n. Nodes with degree zero are called the terminal nodes or leaves. Other nodes, excluding the root, are called nonterminal nodes. The level number of a node is the level number of its father plus one, with the root having zero as its level number. A level in a tree is a set of nodes with the same level number. For an example, in Fig.4.1, consider a tree T in which the level number of D is 2, or we say D is in the level 2 of the tree T. The height of a tree is the maximum level number that a node in a tree can have.

Let the path to a node n be defined as an ordered concatenation of n with all its ancesters, with the root at the beginning position of the concatenation.

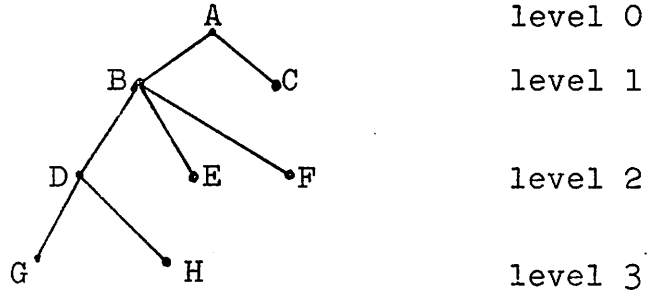


Fig.4.1

For an example, the path to H is ABDH. Within a tree, a node can be labelled with any name, i.e. any alpha-numeric code. In fact two nodes may have the same label. Let us call the label of a node code name of the node. A path of a node can be represented as a concatenation of code names. We call the concatenation of the path to a node the path name of the node. The path name of a node is unique, provided that sons of a node have distinct names. For an example , we have a tree T in which nodes are labelled as shown in Fig. 4.2. There are two nodes in T labelled

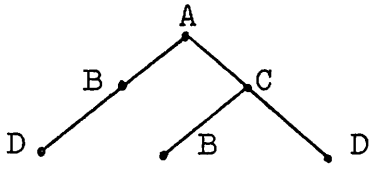


Fig. 4.2

with B, but one has the path name AB, another has ACB. Therefore two nodes with the same code name can be uniquely identified if their path name were used. Similarly, it applies to two nodes with a code name D.

A forest is defined as a collection of zero or more disjoint trees. For example, let T_1, T_2 be two trees as shown in Fig.4.3. $\{ T_1, T_2 \}$ is a forest.

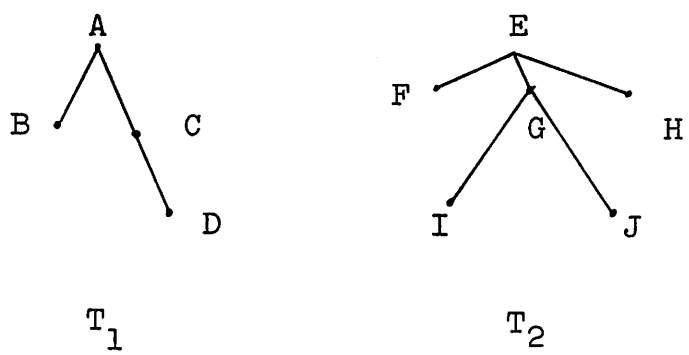


Fig. 4.3

Of course, a forest can be transposed into a tree by connecting all the roots to an extra common node which serves as the root of the created tree.

A <u>binary tree</u> is a finite set of nodes which is either empty or consisting of a root and two disjoint binary trees which are the left and right subtrees of the root.

The differences between trees and binary trees are

(1) A tree is never empty, i.e. it always has at least one node, and each node of a tree can have

0,1,2,...,m sons.

(2) A binary tree can be empty, and each of its node can have 0,1,2 sons; when the number of sons > 0, we distinquish between a left son and a right son.

There is a natural correspondence between forests and binary trees, i.e. every forest can be represented as a binary tree and vise versa. Consider the forest $\{T_1, T_2\}$ shown in Fig.4.3, the corresponding binary tree is obtained by linking root(T_1) with root(T_2) and linking together sons of a node and eliminating all branches to the sons except for the left most one. The corresponding binary tree T_3 for the forest T_1, T_2 is shown in Fig.4.4.

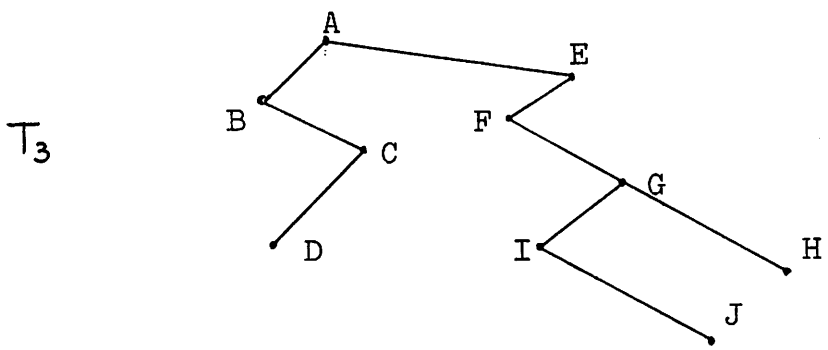


Fig.4.4.

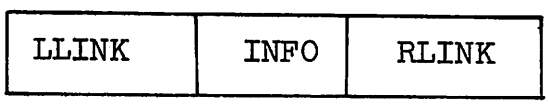
In general, let F = ($T_1,...,T_m$) be a forest. The binary tree B(F) corresponding to F can be obtained formally as follows:

(1) m = 0, B(F) is empty

(2) $m>0$, the root of B(F) is the root of T_1; the left subtree of B(F) is $B(T_{11},\ldots T_{1n})$, where $T_{11},\ldots,T_{1n}$ are subtrees of T_1, the right subtree of B(F) is $B(T_2,\ldots,T_m)$.

Since trees are representable in binary trees and many algorithms in applications are of binary trees nature, it is worthwhile to study binary tree a little further.

By the nature of the definition of a binary tree, there is a simple way to represent a binary tree within a random access computer memory. A node is represented by a set of memory cells, in which there are two fields LLINK and RLINK. LLINK stores the address of the left subtree of the node and RLINK stores the address of right subtree of the node. Within a node there may be another field called INFO, in which information of data or about the tree can be stored. Therefore, in general, a node has the form



There is a variable T which is a pointer to a tree. If the tree is empty, T = 0; otherwise T = address of the tree, and LLINK(T), RLINK(T) denote pointers to the left subtree and right subtree of the root respectively. These rules recursively define a

memory representation of a binary tree within a computer memory. For example, for the binary tree T in Fig.4.5, the corresponding representation in a computer memory is shown in Fig.4.6, where ⏚ in a

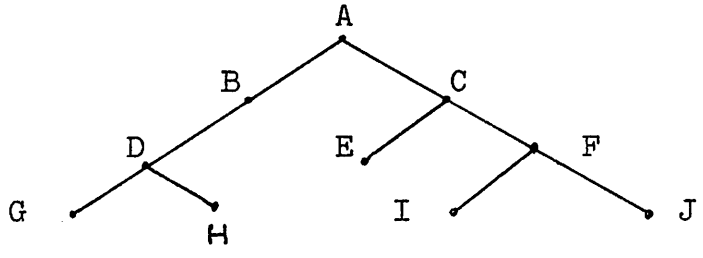


Fig.4.5

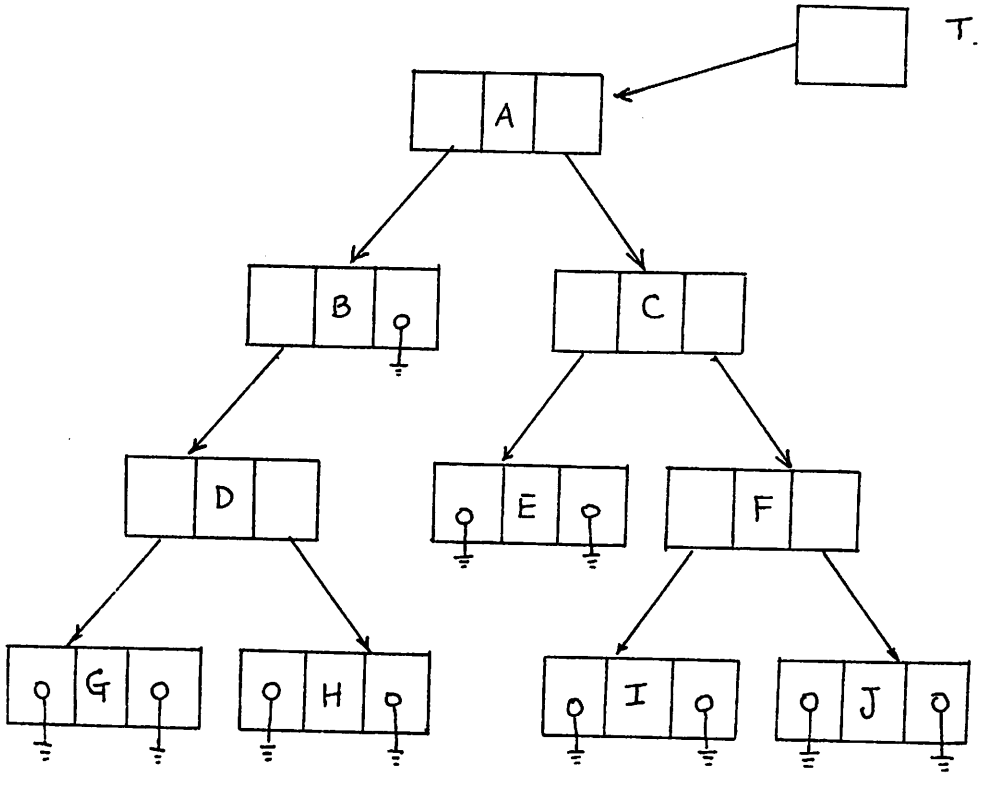


Fig. 4.6

field indicates the corresponding subtree is empty.

The basic algorithm needed for manipulating a binary tree is the algorithm for traversing a binary tree, i.e. examining the nodes of the tree systematically so that every node is traversed only once. A traversing of the whole tree produces a linear ordering of all nodes in a tree. There are three principal ways to traverse a tree, namely, preorder, postorder and enorder which are defined recursively as follows:

Preorder:
Visit the root
Traverse the left subtree.
Traverse the right subtree

Postorder:
Traverse the left subtree
Visit the root
Traverse the right subtree.

Enorder:
Traverse the left subtree
Traverse the right subtree
Visit the root

For the example in Fig.4.6, if we traverse the

tree T by preorder, we are examining the nodes in the following order

ABDGHCEFIJ

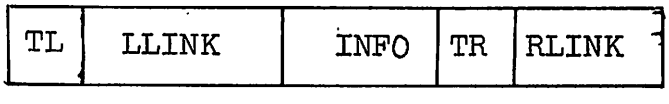
by postorder, we have

GDHBAECIFJ

by enorder, we have

GHDBEIJFCA

Note that fields containing zero are merely a waste of space, because they convey no information other than that the link is terminated. To make use of this extra memory space, the so called "threaded" tree representation is introduced. In this representation, terminal links contain pointers to other parts of the tree, instead of zero. In fact LLINK(P) will contain the precessor of P and RLINK(P) will contain the successor of P, with respect to a certain traversing order. For example, with respect to postorder, a threaded tree representation of the tree T in Fig.4.5 is shown in Fig.4.7, where the thinner lines representing threads. Within a computor, a node is modified to



TL, TR are tags indicating the link fields to which they are attached are threads or regular links. For example,

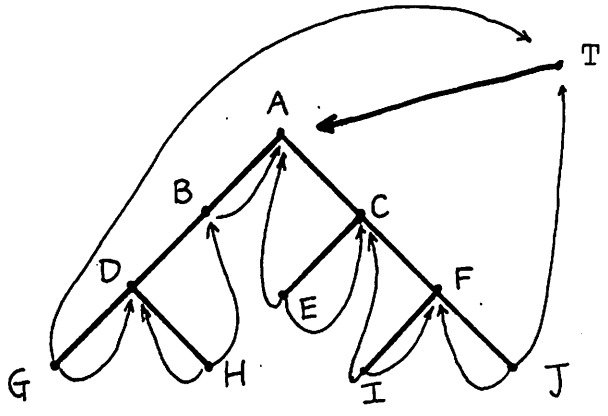


Fig. 4.7

TL = 1 or TR = 1 indicates LLINK or RLINK is a regular link respectively, and TL = 0 or TR = 0 indicates LLINK or RLINK is a thread.

The advantage of a threaded tree is that the traversing algorithms become much simpler, for there is no stack to be maintained in the traversing algorithms. The threads give enough information as to where is the next node to traverse.

IV.2. IMPLEMENTATION OF THE PROPOSED FILE STRUCTURE AS A TREE

IV.2.1. TREE REPRESENTATION:

Let $B(G_0) \subseteq B(G_1) \subseteq \ldots \subseteq B(G_m) = B(\mathcal{L})$ be a sequence of Boolean algebras as defined in Chapter III, $C_0, C_1, \ldots, C_m$ be the sets of atoms of $B(G_0)$, $B(G_1)$, ...,

$B(G_m)$ respectively. Since $\{ C_0, C_1, \ldots, C_m \}$ are a nested set, there is a nature tree structure representation. Let every atom in C_0 be represented by the root of a tree in the forest. Let U be any node in the tree. U represents an atom in some set, say C_i, and sons of U, say $\{ U_1, \ldots, U_p \}$ represent atoms in C_{i+1}.

The procedure for constructing this forest is described as follows:

Let $K = \{ k_1, \ldots, k_n \}$ be the set of keywords. Let $X_1 / X_2 / X_3 / \ldots / X_m$ be a disjoint partition of K, where $X_1 = \{ k_1, \ldots, k_{v_1} \}$, $X_2 = \{ k_{v_1+1}, \ldots, k_{v_2} \}$, $\ldots$, $X_i = \{ k_{v_{i-1}+1}, \ldots, k_{v_i} \}$, $\ldots, X_m = \{ k_{v_{m-1}+1}, \ldots, k_{v_m} = k_m \}$, g_i be a Boolean function, with X_i as the set of variables, mapping from $W^{v_i} \rightarrow W$, for i=1,...,m. Thus, we have $G_0 = \{ g_1, \ldots, g_m \}$, $G_1 = \{ k_1, \ldots, k_{v_1}, g_2, \ldots, g_m \}$, $\ldots$, $G_i = \{ k_1, \ldots, k_{v_i}, g_{i+1}, \ldots, g_m \}$, $\ldots$, $G_m = \{ k_1, \ldots, k_n \}$. Let us assume that $C_0, \ldots, C_m$ are represented in their tabular forms, i.e. atoms are represented as an array of codes which are tuples of "0" and "1". The code name of the root in a tree is the code of an atom in C_0 in the tabular representation. At level i, for i = 1,...,m, of a tree in the forest, a node U is represented by a partial code of an atom

in C_i. Let u be a node representing an atom x in C_i; if x appears in the tabular form of C_i as

$$(y_1,\ldots,y_{v_i},y_{v_{i+1}},\ldots,y_m)$$

then the code name for u is simply

$$(y_{v_{i-1}+1},\ldots,y_{v_i})$$

Furthermore, if u is a descendant of a root T with a code name $(p_1,\ldots,p_m)$, then $g_i(y_{v_{i-1}+1},\ldots,y_{v_i}) = p_i$. Let us consider the example given in Chapter III, $K=(k_1,k_2,k_3,k_4)$; $X_1/X_2 = (k_1,k_2/k_3,k_4)$ is a disjoint partition of K; g_1,g_2 are defined as follows in Table 4.1

x_1	x_2	g_i; i=1,2
0	0	1
0	1	1
1	0	0
1	1	0

Table 4.1

C_0 = (00,01,10,11)

C_1 = (111,010,101,011,100)

C_2 = (1100,0110,1000,0111,0101,1011,1001)

where tuples of 1 and 0 are codes of atoms in the tabular forms of C_i. The forest is represented as follows in Fig. 4.8.

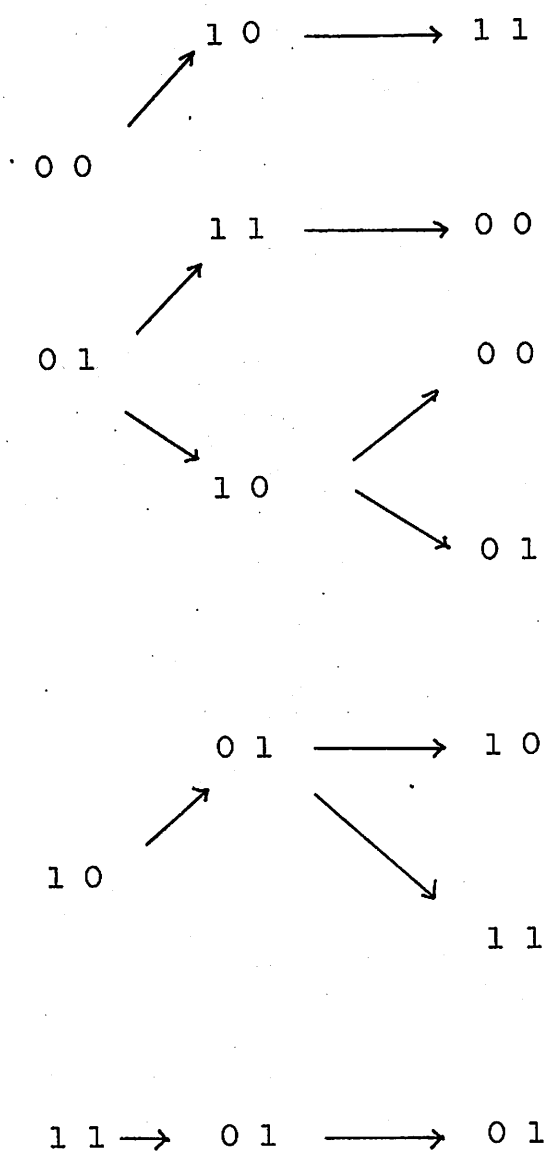


Fig. 4.8

The tree representation of the sets of atoms C_0, $C_1,\ldots,C_m$ can be viewed as a compact representation of the tabular forms of $C_0,C_1,\ldots,C_m$; relationships between C_i are expressed explicitly by branches in the tree and codes of atoms can be decoded from paths in the trees. For the example above, let us use path name to identify nodes in the tree, we can see that (Fig.4.8) there are two sons branching from (0 1) at level 0 of the second tree, namely 0111,0110, which represent two atoms 111,101 in C_1 respectively; 0111 has a son 011100; 0110 has two sons 011000,011001; where (011100, 011000,011001) represent atoms 1100,1000,1001 in C_2. This illustrates the fact that $b_1 \supseteq (a_1,a_3)$, $a_1 \supseteq (c_1)$, $a_3 \supseteq (c_3,c_7)$ as shown in Chapter III.

The retrieval procedure for this tree representation of atom sets is a modification of the retrieval procedure described in Chapter III. For a given query q, a vector $q_m = (p_1,\ldots,p_n)$ is constructed, where $(p_1,\ldots,p_n)$ is in W^{*n}, $W^*=(0,1,*)$ and $p_i=1$ if k_i is in q; $p_i=0$ if $\bar{k}_i$ is in q; $p_i=*$ if neither k_i nor $\bar{k}_i$ is in q. From q_m, obtain vector $q_{mi} = (p_{v_{i-1}+1},\ldots, p_{v_i})$, for i=1,...,m. A vector $q_{m0} = (x_1,\ldots,x_m)$ is constructed from q_m as follows. First of all, q_i, for i=1,...,m, are extended to functions mapping from $W^{*(v_i-v_{i-1})}$ to W^*, in a well defined way. Then let

$x_i = q_i(p_{v_{i-1}+1}, \ldots, p_{v_i})$, for i=1,...,m

After we have constructed $q_{m0}, q_{m1}, \ldots, q_{mm}$, these vectors are used to compare with nodes in the trees. Define the binary operator D between two vectors U,V as in Chapter III. Starting from i=0, at each level i, the value of $D(y_i, q_{mi})$ is computed, where y_i is a code name of a node. If $D(y_i, q_{mi})=0$, then sons of the node with code name y_i are traversed and a subset Y_{i+1} of code names of these sons is obtained, where $Y_{i+1} = \{ y_{i+1} / D(y_{i+1}, q_{mi}) = 0 \}$. Finally , when i=m, a set $Y_m = \{ y_{mm} / D(y_{mm}, q_{mm})=0 \}$ is the set of code names representing atoms in $B(\mathcal{L})$, satisfying q.

Consider the example given in Fig. 4.8. Let $q = k_1 \wedge \bar{k}_3$, then q_m=(1 * 0 *), q_{m1}=(1 *), q_{m2}= (0 *), q_{m0} = (0 1). At level 0, the node 01 satisfies the equation $D(01, q_{m0})=0$; at level 1, two sons (0111),(0110) of 01 , satisfy the equation $D(y_1, q_{m1})=0$, Y=(0111,0110); at level 2, Y_2= (011100, 011000,011001). Since Y_2 is a set of leaves, the procedure stops and atoms of $B(\mathcal{L})$ are decoded as (1100, 1000,1001), which are atoms in $B(\mathcal{L})$ satisfying q.

While update of a file consists of inserting a record into or deleting a record from the file, update of the atom sets in the trees representation involves adding nodes to or deleting nodes from the trees.

An atom of $B(\mathcal{L})$ consists of one or more records; inserting a record to or deleting a record from a file does not always mean adding nodes to or deleting nodes from the tree representation. Adding nodes to the tree is required only if the record to be inserted is contained in an atom that is not yet represented in the tree structure. Deletion of nodes from the tree is required if the record to be deleted is the only record that is contained in an atom or if the whole atom is to be deleted. By update here, we mean update of the atom sets. We shall write an "atom" to mean an atom in $B(\mathcal{L})$.

The procedure for adding an atom to a tree with a height m can be described as follows. From q_m, we obtain $q_{m0}, q_{m1}, \ldots, q_{mm}$ as defined earlier. Consider the following procedure:

(1) i=0

(2) i>m, go to (9)

(3) get the eldest brother x, let y=x

(4) $D(y, q_{mi})=0$, go to (7)

(5) if there are no more brothers, go to (8)

(6) get next younger brother z, let y=z go to (4)

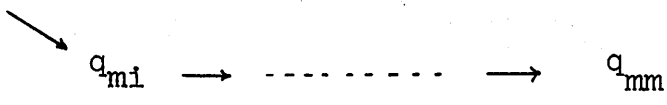
(7) i=i+1 go to (2)

(8) add a partial path with a name $q_{mi}, \ldots, q_{mm}$

stop

(9) stop

This procedure adds an atom to the tree. Note that the value * will never appear in q_m or q_{m0}, therefore at each level i, there is either no node or an unique node y, such that $D(y, q_{mi})=0$. In the case that there is no node y, such that $D(y, q_{mi})=0$, as in step (8), a partial path has to be added to the tree, and it has the form:



For example, consider the forest in Fig. 4.8. Suppose an atom with a code (0011) is to be added to the tree. Let q_m=(0011). Then q_{m0}=10, q_{m1}=00, q_{m2}=11. The third tree in the forest will look like this after insertion, (Fig. 4.9)

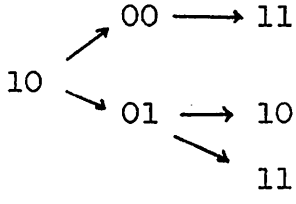


Fig.4.9

where the path to be added is →00 →11 .

When an atom is to be deleted from the atom

sets, a partial path will be deleted from the tree. Let q_m, $q_{m0},\ldots,q_{mm}$ be defined as earlier for an atom to be deleted. The partial path to be deleted from the tree has the form

$$\longrightarrow \quad q_{mi} \longrightarrow q_{m(i+1)} \longrightarrow \quad \cdots \quad \longrightarrow q_{mm}$$

where $q_{m(i+1)}$ is the only son of q_{mk}, $k=i,\ldots,m$.

For example, consider Fig.4.8 and suppose that the atom 1100 in $B(\mathcal{L})$ is to be deleted. Let q_m = (1100). Construct q_{m0} = (01), q_{m1} = (11), q_{m2} = (00). Since the code name T of the root of the second tree satisfies $D(T,q_{m0}) = 0$, we know that (1100) is contained in the second tree which will has the following form after deletion (Fig.4.10):

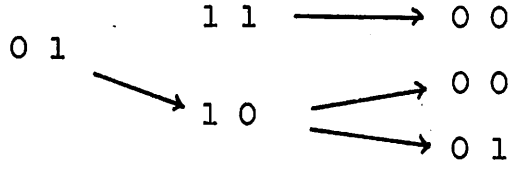


Fig.4.10

where 1 1 ⟶ 0 0 is the partial path that has been deleted. Usually, deletion of a partial path means disconnecting a link from the father of the eldest member in the partial path. The space that was occupied by the partial path is then returned

to the free space list. This is a problem of garbage collection which will not be discussed here.[36]

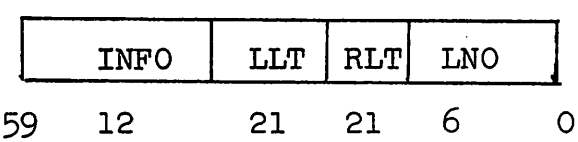
IV.2.2. IMPLEMENTATION PROGRAMS:

The purpose of the programs is to demonstrate feasibility and to serve as a guide to the analysis of the proposed file structure. The programs were written in FORTRAN and COMPASS. A sample file of 1000 records are generated by a pseudo-random number generator, each record being represented by a 100-bit code and each bit representing one keyword. The pseudo-random generator is written so that each record it generates is a concatenation of ten 10-bit pseudo-random numbers generated independently. In a way, it simulates the bit patterns of fields in an actual record. For example, a record in a personel file has a field name AGE . The values for AGE range from 20 to 65, then codes for these values must have some bit patterns. A linear recurrence algorithm is used in writing of the generator. Output of the generator is a set of random numbers punched on cards.

The programs accept the deck of cards as the representation of the set of records. A forest of trees is built up from this representation of records, under the following specifications:

Let $K = \{ k_1,\ldots,k_n \}$, $X_1/X_2/\ldots/X_m$, $g_1,\ldots,g_m$ have the same general meaning as defined in Chapter IV.2.1. In particular, for the purpose of our programs, we choose n=100, m=10, $X_1=(k_1,\ldots,k_{10})$, $X_2=(k_{11},\ldots,k_{20}),\ldots,X_i=k_{m(i-1)+1},\ldots,k_{mi}),\ldots,$ $X_{10}=(k_{91},\ldots,k_{100})$; $g_1,\ldots,g_{10}$ are chosen to be the OR function of $X_1,X_2,\ldots,X_{10}$ respectively. The reason for choosing $g_1,\ldots,g_{10}$ this way is the ease of programming, because the validity of an OR function is relatively easy to verify. The reason for choosing $X_1,\ldots,X_{10}$ to have the same number of elements and to choose $n = m^2$ is that the computer representation of a node in a tree will have a fixed format, another convenience for programming. All these restrictions on K, n, m, g_i are rather artificial. But the programs were written only to illustrate feasibility; hence we made everything simple.

Within a computer, a forest is represented as a threaded binary tree, more precisely, a half threaded binary tree, with the LLINK field of a leaf pointing to its father, and the RLINK field of a leaf storing an atom number. A node, occupying one CDC 6400 computer word, has the following format



where INFO, (12 bits, of which only 10 bits are used), stores the code name of a node; LLT, (21 bits), is the LLINK and LT fields and stores a pointer to its brother, or, if the node is a leaf with a father, stores the negative of the address of its father, or, if the node is a leaf without a father, is filled with 1's; RLT, (21 bits), is the RLINK and RT fields and stores a pointer to its son, or, if the node is a leaf, stores the negative of an atom number of the atom it represents; and, LNO, (6 bits), stores the level number of a node.

Note that negative values in RLT indicate that the corresponding node is a leaf and the negative value in RLT is the atom number of an atom it represents. In reality, RLT may store a pointer to some memory locations in which other information about the atom is stored. A negative value in a LLT indicates that it is a threaded, instead of a regular link. A thread is the negative of a pointer to the father of this node. If the node does not have a father, i.e. a node representing the youngest brother at level 0 in the forest, its LLT has the value of negative zero (all 1's).

For example, consider a forest in Fig.4.11. The binary tree representation for this forest is shown

Fig. 4.11

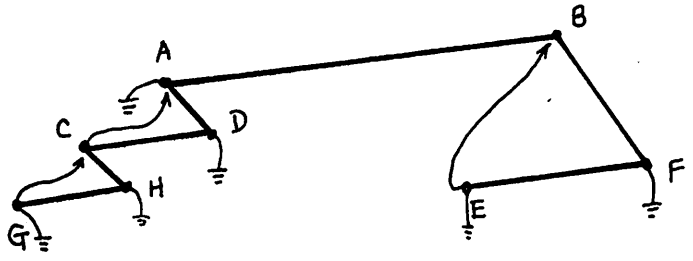


Fig.4.12

In a computer memory, by considering A, B, C, D, E, F, G, H as addresses in the memory, this binary tree has the representation shown in Fig. 4.13, where PT is a pointer to the root of the binary tree, and the thinner lines represent threads. We shall call the binary tree representation within the computer the tree from now on. The tree will be traversed by preorder algorithm, i.e.

Visit the root,

Traverse the right subtree,

Traverse the left subtree.

"Visit the root" means to do something with the root; here it means to compare the code name of a root with a vector q_{mi} as defined earlier for a query. Note

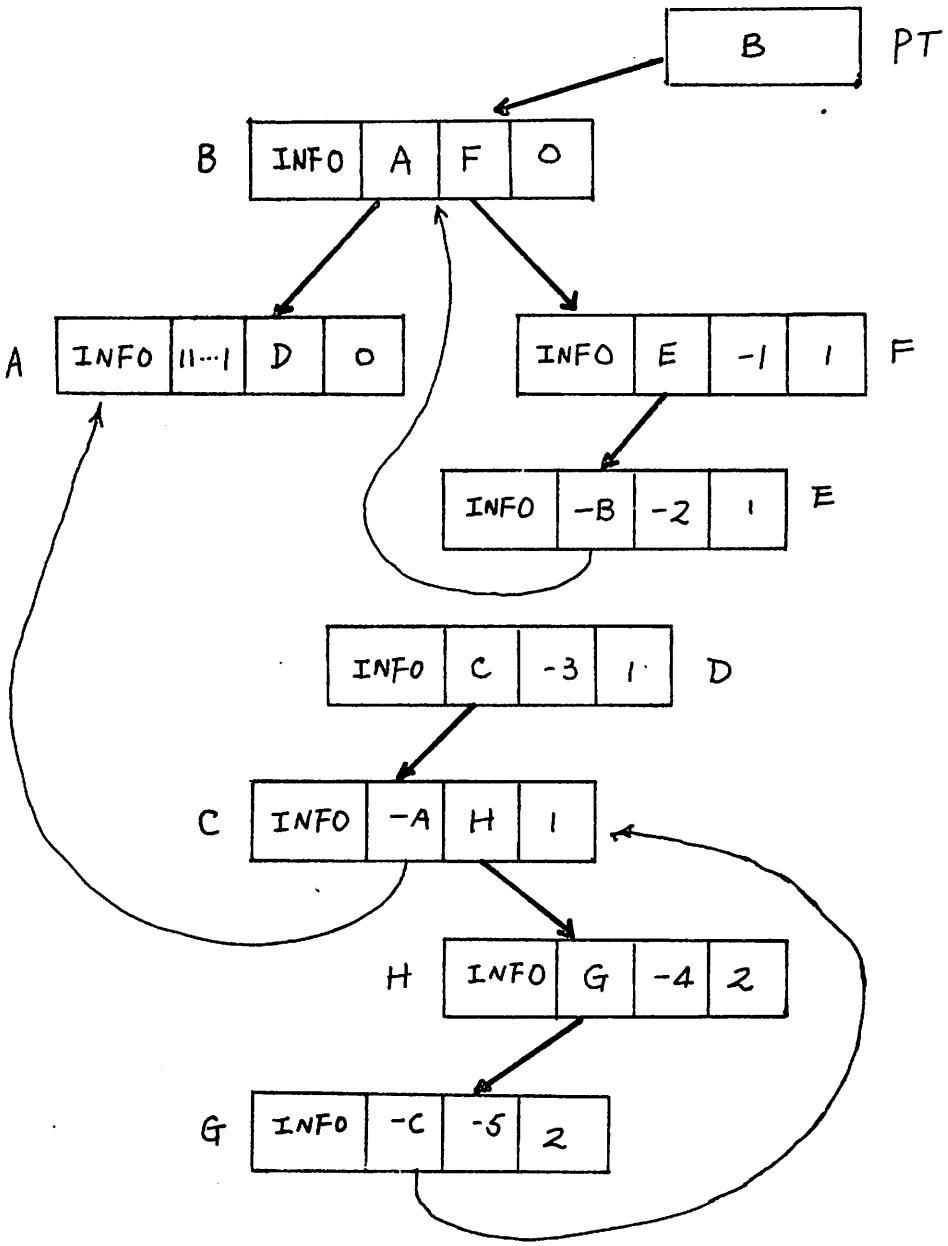

B
PT
B
INFO
A
F
0
A
INFO
11···1
D
0
INFO
E
-1
1
F
INFO
-B
-2
1
E
INFO
C
-3
1
D
C
INFO
-A
H
1
H
INFO
G
-4
2
G
INFO
-C
-5
2

Fig. 4.13

that we traverse the right subtree before traversing the left subtree, which is different from the definition of preorder given in Section IV.1. However, it should be clear that it is just a matter of reordering of brothers in a forest.

Our program consists essentially of eight subprograms, namely, TREE, CONVERT, DETREE, RETRIEV, ADDBIT, RETRI, MASH and MATCH. Relationship among these programs is shown in Fig.4.14, where a link represents a call from one program to another.

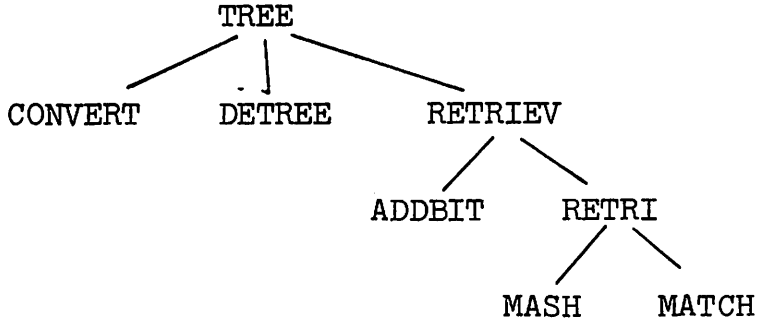


Fig. 4.14

These programs first convert a set of records into the tree, then they process a query and print out all those atoms that satisfy the query. We assume that each atom in B($\mathcal{L}$) contains one record for the sake of simplicity. Therefore, the set of records is identical with the set of atoms in B($\mathcal{L}$). In practice, each

atom may contain more than one record. A pointer to the set of records belonging to that atom must then be stored in the leaf representing that atom, instead of the record number being stored as we did.

The function of each individual program will be described as follows:

TREE: It is the main program which accepts the set of atoms punched on cards; calls CONVERT to convert each atom into an appropriate form; calls DETREE to construct the tree; finally calls RETRIEV to read a query and retrieve from the tree all atoms satisfying that query. This main program is written in FORTRAN.

CONVERT: It converts each atom into a vector q_{m0} defined earlier, i.e. for an atom $(X_1,\ldots,X_{100})$, construct a vector $q_{m0} = (P_1,\ldots,P_{10})$ such that $P_i=1$, if $g_i(X_{m(i-1)+1},\ldots,X_{mi}) = V(X_{m(i-1)+1},\ldots,X_{mi}) = 1$ or equivalently, if there exist a k in $\{ m(i-1)+1,\ldots,mi\}$ such that $X_k=1$; $P_i=0$, if $g_i(X_{m(i-1)+1},\ldots,X_{mi}) = V(X_{m(i-1)+1},\ldots,X_{mi}) = 0$ or equivalently, if for all k in $\{ m(i-1)+1,\ldots,mi \} \ni X_k=0$.

Functionally, CONVERT determines codes for nodes at level 0 of trees in the forest. CONVERT is written in FORTRAN.

DETREE: Constructs the tree from the set of atoms. For each atom, DETREE determine what part of the tree

the atom belongs to. A vector q_{m0} is obtained from the output of CONVERT. q_{mi}, for i=1,...,m are obtained by left shifting 10 bits of the vector q_m of each atom consecutively into a register M. q_{mi}, i=0,...,m are compared one after another with code names in the tree to determine where in the tree q_{mi} should be stored. DETREE is written in COMPASS.

RETRIEV: Calls ADDBIT to translate a query q from query language to a vector form that is appropriate for processing. And then calls RETRI to traverse the tree and to retrieve atoms satisfying q from the tree. RETRIEV is written in FORTRAN.

ADDBIT: The query language we use here is very simple. To specify a query, a sequence of integers is used to specify which keywords or complement of keywords are in the query. All queries are in conjunctive forms. Negative integers are used to specify the complements of keywords. For example, (-5 6 9) means $\bar{k}_5 \wedge k_6 \wedge k_9$. Instead of using the symbol *, ADDBIT converts a query q in a form of a sequence of integers into three vectors: $q_{m0}=(z_1,\ldots,z_m)$, where m=10, and $z_i=0$ if there exists a j in $\{ m(i-1)+1,\ldots,mi \}$ such that k_j is in q; $P_m=(x_1,\ldots,x_{100})$, $q_m=(y_1,\ldots,y_{100})$, where $x_i=0$ if k_i is in q, $y_i=0$ if $\bar{k}_i$ is in q. For example, if q=(-5 6 9), we will have q_{m0}=(011...1), P_m=(11111011011...1),

q_m=(1111011...1). The reason for doing this will be given when MATCH is described. ADDBIT is written in COMPASS.

RETRI: The tree is traversed in preorder by RETRI. When the root of a subtree is being visited, the code name of the root is compared with a vector which is a result from a call to MASH. A comparison between a code name and a vector is done by a call to MATCH. The traversing process stops when the node being visited has a LLT containing a negative zero(11...1). It means that there are no more trees in the forest to be traversed. RETRI is written in COMPASS.

MASH: Since P_m, q_m are vectors of 100 bits, each needs two CDC computer words to be stored. P_m is stored in IQUERY(1) and IQUERY(2). q_m is stored in IQUERY(3) and IQUERY(4). MASH leftshifts P_m,q_m 10 bits each time consecutively into MM(1) and MM(2) respectively. MM(1) and MM(2) are used in MATCH to compare with code names in the tree. MASH is written in COMPASS.

MATCH: MATCH performs a matching procedure between a code name INFO(n) of a node n and MM(1), MM(2). Let $q=(P_1,P_2,\ldots,P_{100})$ be a vector representation of a query q, as described in Section IV.2.1, i.e. where P_i=1, if k_i is in q; P_i=0 if $\overline{k}_i$ is in q; P_i=* if neither

k_i nor $\overline{k}_i$ is in q. Let $q_{mi}=(P_{m(i-1)+1},\ldots,P_{mi})$.

Claim: $D(INFO(n),q_{mi})=0$ if $MM(1) \bigvee INFO(n) = (11\ldots1)$ and $MM(2) \bigvee \overline{INFO(n)} = (11\ldots1)$.

Proof of the claim:

Let $INFO(n) = (u_1,\ldots,u_m)$, $u_i=0$ or 1. By definition $d(INFO(n), q_{mi}) = 0$ if $u_j - P_{m(i-1)+j} = 0$ for all $j=1,\ldots,m$. Also by definition $MM(1) = (x_{m(i-1)+1}, \ldots,x_{mi})$, $MM(2) = (y_{m(i-1)+1},\ldots,y_{mi})$ where

$$x_{m(i-1)+j} = 0 \quad \text{iff} \quad P_{m(i-1)+j} = 1$$
$$= 1 \quad \text{iff} \quad P_{m(i-1)+j} = 0 \text{ or } *$$
$$y_{m(i-1)+j} = 0 \quad \text{iff} \quad P_{m(i-1)+j} = 0$$
$$= 1 \quad \text{iff} \quad P_{m(i-1)+j} = 1 \text{ or } *$$

Suppose $MM(1) \bigvee INFO(n) \neq (11\ldots1)$ or $MM(2) \bigvee \overline{INFO(n)} \neq (11\ldots)$. $MM(1) \bigvee INFO(n) \neq (11\ldots1)$ implies that there is a j such that $u_j \bigvee x_{m(i-1)+j} = 0$, which implies $u_j=0$ and $x_{m(i-1)+j}=0$, implies $u_j=0$ and $P_{m(i-1)+j}=1$, implies $u_j - P_{m(i-1)+j} \neq 0$, implies $D(INFO(n),q_{mi}) \neq 0$, contradicting to the fact that $D(INFO(n),q_{mi}) = 0$.

$MM(2) \bigvee \overline{INFO(n)} \neq (11\ldots1)$, implies that there is a j such that $\overline{u}_j \bigvee y_{m(i-1)+j}=0$, implies $\overline{u}_j=0$ and $y_{m(i-1)+j}=0$, implies that $u_j=1$ and $P_{m(i-1)+j}=0$, implies $u_j - P_{m(i-1)+j} \neq 0$, implies $D(INFO(n),q_{mi}) \neq 0$, also a contradiction to $D(INFO(n),q_{mi}) = 0$. Q.E.D.

By using this claim we can process the query without using the symbol *. MATCH is written in COMPASS.

While the statements of these programs can be found in the Appendix, the algorithms for these programs will be given as follows:

TREE:

0. Read N, K. Allocate space for TR.
   (N is the number of cards, K is the number of bits in a code name, TR is the address of the tree.)
1. PT$\leftarrow$ TR. (PT stores a pointer to a root.)
2. Read Ith card, and I+1th card into B(I) and B(I+1).
3. Call CONVERT (B,I,M).
   (B is a fuffer, storing an atom; I, number of cards being read; M, code name of a node.)
4. NL=1, J=0. (NL, level number to be stored in LNO, but actually, NL is a number greater than the level number by 1, i.e. at level 0, NL=1; J is a counter for number of leftshifts.)
5. I_3 = (I+1)/2. (Each atom occupying two cards, I_3 is an atom number.)
6. Call DETREE (M, NL, TR, PT, I_3).
7. Leftshift B, (J+1)*10 bits into M, MASK left most 10 bits of M, put it into M.
8. J 10, Go to 10.
9. NL =NL+1, J=J+1 Go to 4.
10. I $>$ N. Go to 12. (All card read?)

11. I = I + 2. Go to 2. (Read the next two cards.)
12. Print TR. (Print the whole tree.)
13. Call RETRIEV (TR, KEY, NK).
    (KEY is an array containing keyword numbers or the negatives of keyword numbers in a query, NK is the number of keywords appearing in a query.)

CONVERT (B, I, M):

0. J=0, JA=0. (J is a counter for the number of leftshifts, JA stores the bit position that has to be a "1" in M.)
1. IA ⟵ [B(I)]. (Content of B go into IA.)
2. Leftshift (zero filled) IA, J*10 bits into IA.
3. MASK left most 10 bits of IA into IT.
4. IT=0 Go to 6.
5. Set p^{th} bit of M to "1", $p=Mod_{10}[JA*10] + 1$.
6. J=J+1, JA=JA+1.
7. J=6 Go to 10.
8. JA > 10. Go to 11. (Pattern of left most 10 bits of M has been determined.)
9. Go to 2.
10. I=I+1 Go to 1.
11. Return.

DETREE (M, NL, TR, PT, RN):

Notations: [A] means the content of a variable A.

p ⟵ B means address of B being put into p; p ⟵ [B] means content of B being put into p; F(p) means the field F in a node p, for example, INFO(p) means the INFO(p) in P; and -0 = (11...1).

1. p ⟵ TR. (p is a pointer to the root of a subtree.)
2. [[p]] = 0 Go to 4. (The tree is empty.)
3. Go to 6. (If the tree is not empty.)
4. INFO([p]) ⟵ [M] ; [p] = TR, LLT ⟵ -0, RLT ⟵ -[RN] , LNO ⟵ [NL] , PT ⟵ [PT] + 1, Return.
   (PT is a pointer to an available space; -0 will be propagated to the brother of p; RN is an atom number propagated to the son of p.)
5. LNO ⟵ [NL] , RLT ⟵ -[RN] , LLT ⟵ -([PT] -1), PT ⟵ [PT] + 1, Return. (Address of its father being put into LLT.)
6. [LNO] = [NL] Go to 11. (The level number of a node is matched with that of the on-coming atom.)
7. RLT ≠ -, Go to 9. (If it has a son.)
8. RN ⟵ [RLT] , RLT ⟵ [PT] , p ⟵ [PT] Go to 2.
9. NEXT = 0, p ⟵ ([PT] - 1) Go to 2.
   (NEXT is a pointer to the next memory address to be examined; if NEXT = 0, let p be the

address of a node that is just implemented before this call of DETREE.)

10. Go to 17. (NEXT ≠ 0.)
11. M ≠ INFO([p]), Go to 14. (Code name not existed yet.)
12. RLT([p]) = -, Return. (Code name already existed.)
13. Go to 17.
14. [LLT] ≠ - Go to 16. (Not the youngest brother.)
15. TEMP ⟵ [LLT] , LLT ⟵ [PT] , p ⟵ [PT] , INFO([p]) ⟵ [M] , LNO(p) ⟵ [NL] , LLT(p) ⟵ [TEMP] , RLT(p) ⟵ - [RN] , NEXT ⟵ [PT] , PT ⟵ [PT] + 1, Return. (A new brother node is implemented, NEXT contains address of a node next to be visited.)
16. p ⟵ [LLT] Go to 11. (Go back to check the code name of the next younger brother.)
17. p ⟵ [NEXT] , NEXT ⟵ 0, Go to 2. (NEXT gives the next address to be visited without traversing any branch.)
18. NEXT ⟵ [RLT([p])] , Return.

RETRIEV:

1. Read NK. (NK is the number of keywords in a query.)
2. Read KEY(I), I=1, NK. (KEY(I) stores the keyword numbers.)

3. $k_1=1$, $k_2=k_1+9$. (To form a disjoint partition X_1/X for the keyword set K, $K = X_1 \cup X$, and $X_1 = (k_{K_1}, \ldots, k_{K_2})$.)
4. KEY(I) $>$ 0, Go to 6.
5. Go to 2.
6. $k_1 \leqslant$ KEY(I) $\leqslant k_2$, Go to 10. (k_j in X_1, j=KEY(I).)
7. $k_1=k_1+10$. (To form X_2.)
8. $k_1 >$ 100, Go to 13.
9. $k_2=k_1+9$ Go to 6.
10. NBIT = Mod_{10}(KEY(I))+1.
11. If there is some I_1, such that KEY(I_1) KEY(I) and $k_1 \leqslant$ KEY(I_1) $\leqslant k_2$, Go to 2.
12. Call ADDBIT (M, NBIT). (To construct a vector M representing q_{m0}.)
13. Continue.
14. IQERY(I) $\leftarrow$ -0, I=1,...,4. (IQUERY(I), I=1,2, contains information about which keywords are in the query; IQUERY(I), I=3,4, contains information about complements of which keywords are in the query.)
15. Call RETRI (TR, IQUERY, M). (To traverse the tree.)
16. End.

ADDBIT (X, N). (X is a vector of "0" and "1", and

N is an integer.)

1. Set N^{th} bit of X to zero.

RETRI (TR, IQUERY, M).

0. J ← 0.

1. PT ← TR.

2. INFO([p]) = M? Yes, Go to 5. (If M matches with the code name of the root of a subtree, go to its son.)

3. LLT([PT]) = -0? If no, Go to 4, else stop.

4. PT ←—LLT([PT]) Go to 2. (Visit its brother.)

5. PT ← RLT([PT]).

6. JJ=J*k, NLQ=J+2. (Traversing the tree at level 1.)

7. Call MASH(JJ,IQUERY, MM).

8. If MM(1)=-0 and MM(2)=-0 Go to 16. (If there is no keyword or complement of keyword of X_i in a query, let i=i+1.)

9. Call MATCH (PT, MM, NLQ).

10. If match Go to 16, else if NLQ≠LNO Go to 19.

11. LLT([PT]) = - Go to 13.

12. PT ← LLT([PT]) Go to 9.

13. PT ← -LLT([PT]), J ←LBO([PT])-2, if J=-1, let J=0, Go to 3.

14. If LLT([PT]) = - Go to 13.

15. PT← LL([PT]) Go to 6. (Visit its brother.)

16. If RLT([PT]) > 0 Go to 18.

17. Print -RLT([PT]) Go to 11. (Print atom number.)
18. PT ← RLT([PT]), J=J+1 Go to 6.
19. J=J+1 Go to 6.

MASH (JJ, INQUERY, MM):

1. If JJ=0, TEMP(1) ← [IQERY(1)] , TEMP(2) ← [IQUERY(3)], Go to 5.
2. If JJ ⩾ 60 Go to 6.
3. TEMP(1) ←[IQUERY(1)] , TEMP(2) ← [IQUERY(3)] .
4. Left shift TEMP(1) JJ bits → TEMP(1)
   Left shift TEMP(2) JJ bits → TEMP(2).
5. Mask the left most 10 bits of TEMP(1), TEMP(2) into MM(1), MM(2) Return.
6. If JJ=60 Go to 9.
7. JJ=JJ-60.
8. TEMP(1) ← [IQUERY(2)], TEMP(2) ← [IQUERY(4)] Go to 4.
9. TEMP(1) ← [IQUERY(2)], TEMP(2) ← [IQUERY(4)] Go to 5.

MATCH (PT, MM, NLQ):

1. If LNO([PT]) ≠ NLQ Go to 9.
2. If MM(1) = -0 Go to 6.
3. MM(1) .OR. INFO([PT]) → MM(1).
4. If MM(1)=-0 Go to 6. (Indicates keywords is matched.)

5. MN=+1 Return.
6. MM(2) .OR. -INFO([PT]) → MM(2).
7. If MM(2)=-0, then MN ← -0 Return.
   (Complements of keywords also matched, return.)
8. MN=+1 Return. (No match.)
9. MN=0 Return. ([NLQ] ≠ [LNO] , wrong level.)

IV.3. STORAGE REQUIREMENT AND RETRIEVAL TIME - A COMPARISON TO AN INVERTED FILE:

The storage requirement of the tree is directly proportional to the total number of nodes and the size, being the number of bits, of each node in the tree. In the most general case, the model for our file structure may have many sequences of Boolean algebras. For example, if we have 1000 keywords, we can partition them into 100 groups, each group has 10 keywords, and then partition these 100 groups into 10 groups, each new group contains 10 old groups. In this way, we are actually building two subsequences of algebras, say $B(F_0), B(F_1), \ldots B(F_9)$; $B(G_0), B(G_1), \ldots, B(G_{100})$, such that $B(F_0) \subseteq B(F_1) \subseteq \ldots \subseteq B(F_9) \subseteq B(G_0) \ldots \subseteq B(G_{100})$, where $F_0, F_1, \ldots F_9, G_0, \ldots, G_{100}$ are sets of generators. Let $G_0 = (g_1, \ldots, g_{100})$, then $F_9 = (g_1, \ldots, g_{90}, f_{10})$, where f_{10} is a function of

$g_{91},\ldots,g_{100}$; $F_i=(g_1,\ldots g_{10(i-1)}, f_i,\ldots,f_{10})$, where f_i is a function of $g_{10(i-1)+1},\ldots,g_{10i}$. Corresponding to each subsequence of algebras, there is, in the tree representation, a set of nodes which are representing atoms of algebras in the subsequences. Let us call such a set of nodes a layer of a tree. In our implementation example, there is only one subsequence of algebras, therefore the tree has only one layer.

In general, let n be the number of keywords, m be the number of bits in the code name of a node, and $n=m^k$; then the height h of a tree can be calculated as follows:

$$h = 1+m+m^2+\ldots+m^{k-1} = \frac{m^k - 1}{m - 1} = \frac{n - 1}{m - 1}$$

Let u_i be the average number of nodes at level i. u_i is defined as

$$u_i = \frac{\text{total number of nodes at level i}}{\text{number of the sets of sons of nodes at level i-1.}}$$

As the height of a tree is defined as the maximum level number among all nodes in a tree, the total number N of nodes in a tree can be calculated as follows:

$$N = \sum_{j=1}^{h} \prod_{i=1}^{j} u_i$$

If $u_i=u$, a constant, at all levels, then

$$N = \sum_{j=1}^{h} u^j = u \frac{u^k - 1}{u - 1} \cong \frac{uR}{u - 1}$$

where $R = u^k$ the number of leaves that are atoms in $B(\mathcal{L})$. Let M be the size of a node in terms of bits which include bits for the code names, links and other data. Then, the total storage requirement for the tree is

$$S = MN = M \cdot \frac{uR}{u - 1}$$

The storage requirement S may be much smaller than that of storing the atoms of $B(\mathcal{L})$ in a tabular form, if the overhead ratio, M/m, in each node is close to 1. Let C be the number of bits required to code each atom, the total storage requirement S_1 for storing the tabular form of atoms is

$$S_1 = CR$$

and

$$\frac{S}{S_1} = \frac{M}{C} \cdot \frac{u}{u - 1} \; ; \qquad S < S_1 \quad \text{if } \frac{M}{C} \frac{u}{u-1} < 1.$$

If h and R are fixed, the worst case happens when every node has only one son, i.e. the forest becomes R linear lists, then

$$S = h\, R\, M$$

Again, if $M/m \simeq 1$, implies $Mh=C$, then $S=CR$. Certainly, if $M > m$, then $S > S_1$. For example, if h=11, m=10, M=60,

R=1000, as in our case, S=11x1000x60 = $66x10^4$ bits, S_1 = 100x1000 = $10x10^4$ bits. S = $6.6S_1$ is the worst case.

In order to save storage space,we can store the node with its only son, if it is the case, consecutively, without using a pointer. In this way, the spaces for storing links are saved, but a node will have variable size. Let us define an X-tree as a rooted tree in which every node (except the leaves) has at least two sons. For a given R and a given h, a X-tree exists if $h+1 \leqslant R$; a binary tree exists if $h+1 \leqslant R \leqslant 2^h$. An upper bound for the maximum storage requirement for a X-tree with R leaves, if the X-tree exists, can be shown to be 2R-1 in the following Theorem.

Theorem: The number of nodes including the root and leaves in a X-tree (including binary X-tree) with a fixed number R of leaves is smaller than 2R-1.

In order to prove this Theorem, we need the following Lemmas:

Lemma 1: A binary X-tree with R leaves has 2R-1 nodes, including the root and leaves.

Let us denote an X-tree with R leaves by t(R). The number of nodes of t(R) is denoted by Nt(R). We are to show that for a binary X-tree $t_b(R)$, $Nt_b(R)$ = 2R-1.

Proof: It can be proven by induction on R.

(1) Let R=2, $Nt_b(R)=3$, as shown in Fig.4.15.

$Nt_b(2) = 3$

Fig.4.15

(2) Suppose $Nt_b(R) = 2R - 1$, for $R \leqslant m$;

(3) Let $R = m + 1$.

Let us remove two brother leaves from $t_b(m+1)$. The result is a binary X-tree with m leaves, i.e. $t_b(m)$. It is clear that $Nt_b(m+1) - 2 = Nt_b(m)$. Thus, $Nt_b(m+1) = (2R-1) + 2 = 2(R+1) - 1$ Q.E.D.

Lemma 2: For a given positive integer R, a binary X-tree is a X-tree that has the maximum number of nodes, among all X-trees with R leaves.

Proof: It can be proven by showing that for any non-binary X-tree $t(R)$ with R leaves, there exists a binary X-tree $t_b(R)$, such that

$$Nt_b(R) > Nt(R).$$

Let $t(R)$ be an abitrary non-binary X-tree with R leaves. In the tree $t(R)$, let S be the set of all those nodes that have more than two leaves as their sons. Let us convert $t(R)$ into a binary X-tree with R leaves by removing one son of a node in S each time and connecting two leaves to another leaf of $t(R)$. At each step, the intermediate result is a X-tree $t_1(R)$, and $Nt_1(R) = Nt(R+1)$. Repeat this process until

S is empty. Now we have a binary X-tree $t_b(R)$, and $Nt_b(R) > Nt(R)$. Q.E.D.

Now the proof of the Theorem is obvious from the results of Lemma 1 and Lemma 2.

Proof of the Theorem:

For any given non-binary X-tree t(R), by Lemma 2, there exists a binary X-tree $t_b(R)$, such that $Nt_b(R) > Nt(R)$, and by Lemma 1, $Nt_b(R) = 2R - 1$. Thus, $Nt(R) < 2R - 1$. Q.E.D.

The query processing time in a file structure is dependent on how a file is logically organized and how it is physically stored. This query processing time consists of the processing time of the central processing unit (CPU) of a computer and the actual record accessing time of a random access device, say a disk. When a query is processed, physical records of a disk are accessed into core in which the structure of a file is searched. In our case, we assume that the search time of the tree for a query is proportional to the number of links in the tree to be traversed. Let us call the set of all sons of a node the filial set of the node. Let u_i be the average number of sons in a filial set of a node at level i as defined earlier. Let r be the ratio of the number of nodes satisfying a query within a

filial set versus the total number of nodes in a filial set. The search time t in terms of number of links to be traversed, can be determined as follows. At level i, there will be $r^{i-1}u_{i-1}u_i$ nodes that satisfying the query; therefore there are $r^{i-1}u_{i-1}u_i$ links to be traversed. Thus, the total expected number of links to be traversed in a h level tree is

$$t = \sum_{i=1}^{h} r^{i-1} \prod_{j=1}^{i} u_j$$

In the case of u_i=u a constant

$$t = \sum_{i=1}^{h} r^{i-1} u^i = u \cdot \frac{r^h u^h - 1}{r u - 1}$$

when $ru \longrightarrow 1$, $t \longrightarrow uh$. The limiting case occurs when there is only one atom in $B(\mathcal{L})$ satisfying the query; at each level i there is only one node to be traversed For many queries, the tree is searched without traversing down to the leaves level. For example, if a query consists of keywords from $X_1, \ldots X_i$ of a disjoint partition $X_1 / X_2 / \ldots / X_m$ of the keyword set, then the tree only has to be traversed down to level i, for all descendants of the satisfying nodes at level i are retrieved as the answer to the query. Thus, we have

$$t = u \cdot \frac{u^i \cdot r^i - 1}{u \cdot r - 1}$$

If we consider all these factors, the average retrieval time over all queries will be smaller than

$$t = u \cdot \frac{r^h \cdot u^h - 1}{r \cdot u - 1} .$$

This tree representation of the proposed file structure can be viewed as a file organization that allows retrieval by a partial key, with concatenation of codes of keywords in a record being the primary key. The level 0 in the forest can be viewed as an index to the rest of the forest. Indexes of this kind are not ordered by the ordering of the primary keys but are calculated by a evaluation of some Boolean functions of keywords. It is conjectured that the query processing time is more uniform than other secondary keys retrieval allowable file structure, say inverted file. The uniformity is justified by the fact that for those queries involving few keywords, few numbers of levels of the tree have to be searched for those queries involving many keywords, few numbers of trees in the forest have to be traversed. Further research is in order to justify this hypothesis.

In an inverted file, the CPU processing time is proportional to the number of inverted lists to be intersected and the number of elements in each list. Suppose the intersection of two lists L_1, L_2 is obtained by two way merge of these two lists. Let L_1 have r elements and L_2 have s elements; the minimum number of comparisons is the minimum of two numbers r and s. The maximum number of comparisons has to be performed as r+s-1. If there are k keywords in a query, then there are k lists to be operated on. Let $L_1,\ldots,L_k$ denote these k lists. Let us order these lists so that L_1 is the shortest list among them. Assuming the intersection of these k lists is obtained by repeatedly intersecting of two lists L_i and N_{i-2}, $N_{i-2} = L_{i-1} \cap N_{i-3}$. The final result is a list N_{k-1} which will be a list of addresses of all the records satisfying the query. N_{k-1} is the shortest list among all lists being operated on. The minimum of the maximum number of comparisons it needs to process a query with k keywords is calculated by

$$t_1 = L_1 + (k - 2)\, N_{k-1}$$

Note that t_1 is a function of the number of keywords in a query, the length of the shortest list and the number of "hits" of each intersection.

For an example, consider a file with 10^6 records,

10^3 keywords. Assuming 20 keywords per record, the average number of addresses in an inverted keyword list can be calculated by

$$L = \frac{20 \times 10^6}{1000} = 2 \times 10^4$$

Suppose a query contains 10 keywords and assume arbitrarily that the minimum number of "hits" per intersection is 100. Then, the minimum maximum number of comparison is

$$t_1 = L + (k - 2) \cdot N_{k-1} = 2x10^4 + (8)x100 \simeq 2x10^4.$$

On the other hand, assuming on the average each filial set has 10 nodes, i.e. u=10, and h=6, r=0.3, then the processing time for the query in the proposed file structure is

$$t = 10 \frac{3^6}{3 - 1} = 5 \times 3^6 = 3645$$

Note that u,r are very critical parameters. For example, let r=0.5, u=10, then $t=3.75x10^4$. On the other hand, let r=0.5 and u=4, then t=256.

Suppose we use disk as the storage device; once the characteristic of disks is known, we can compare the access time for the proposed file structure with that of the inverted file. As we mentioned earlier, level 0 of the forest can be viewed as a index to the rest of the forest; we can store level 0 in core and the rest of the forest can be stored level by level

or by preorder on disk. The basic characteristics concerning about disks are the head position time, T_p, the latency time, T_l, the track read time, T_r, which is equal to the time for one rotation of the disk. T_l is assumed to be $T_r/2$. Typical value of these timings for a disk say an IBM 2311, are

$$T_p = 75 \text{ ns}, \quad T_r = 25 \text{ ns}, \quad T_l = 12.5 \text{ ns}.$$

A cylinder of a disk is defined as tracks that can be read at one position of the head. Suppose each tree in the forest is stored in one cylinder. Assuming at level 0, on the average, there are ru nodes satisfying a query. We can formulate the access time as

$$\begin{aligned} T &= ruT_p + 1.5T_r + 2(n-1)T_r \\ &= ruT_p + (2n - 0.5)T_r \end{aligned}$$

where n is the number of tracks to be read and is dependent on the size of a tree. The second term in the formula is the sum of one track read time plus one latency time. The third term is the sum of (n-1) track read time plus one rotation loss for the CPU processing time after every reading of one track for (n-1) tracks. If the whole set of keywords is chosen for the set of generators for $B(\mathcal{L})$ and if every record is solely defined by a subset of keywords, then all those records satisfying a query can be decoded

from the tree structure and T is the time for processing the query.

In case of inverted file, the processing time consists of three parts, T_D, T_I and T_A.[11] T_D is the decoding time for the index for the inverted lists; T_I is the processing time for the inverted lists; and T_A is the access time for accessing the satisfying records. Let us define some symbols:

L	average length of a list.
A	number of file record addresses per physical record (say a track).
N_t	number of keywords in a query.
p	ratio of number of satisfied records in a list over total number in a list.
T_{mD}	decoding time for a m-level index decoding tree.

The formulation for T_{mD} with first level of the index decoder stored in core is

$$T_{mD} = t_p + 1.5t_R + 2(m-2)t_R \qquad m > 1$$

The total decoding time T_D for a query with N_t keywords is $T_D = N_t T_{mD}$

Before finding the intersection of N_t lists in core, we must transfer $L/A\ N_t$ physical records (say tracks) from disk into core. Therefore the processing time T_I for N_t inverted list is

$$T_I = N_t \cdot \lceil L/A \rceil \cdot (T_p + 1.5\ T_r)$$

Finally pL satisfying records must be accessed from the disk. We have

$$T_A = pL\ (T_p + 1.5\ T_r)$$

The total query processing time is

$$\begin{aligned} T_1 &= T_D + T_I + T_A \\ &= T_p + (2m-2.5)T_r + N_t\ L/A\ (T_p + 1.5\ T_r) \\ &\quad + pL\ (T_p + 1.5\ T_r) \\ &= (1 + N_t\ L/A + pL)T_p + (2m + 1.5\ N_t\ L/A \\ &\quad + 1.5pL - 2.5)T_r \end{aligned}$$

Assuming n=10, i.e. 10 tracks per cylinder, consider the previous example, ru=3, $T = 10T_p + (20-0.5)T_r$. For A=500, $L=2\times10^4$, L/A=40, N_t=10, m=3 and pL=100, T_1 can be calculated as

$$\begin{aligned} T_1 &= (1 + 10\times40 + 100)T_p + (6 + 600 + 150 - 2.5)T_r \\ &= 500\ T_p + 750\ T_r \end{aligned}$$

For the purpose of comparison, we shall derive the storage requirement for the inverted file. The total storage requirement for the inverted file also consists of three parts: S_D, S_I, and S_A.

S_D is the storage requirement for the index decoder.

S_I is the storage requirement for the inverted lists.

S_A is the storage for the data file.

Suppose that the decoder has h levels, the branching factor is m, i.e. each node has m sons. Let N be

the number of keywords, $N=m^h$.

The size of the decoding tree can be calculated by

$$D = m \frac{m^h - 1}{m - 1} \leq \frac{m}{m - 1} N$$

We are only interested in how much information we have to store in terms of the number of bits to represent the collection of data. The physical size of storage we need , say the number of tracks, can be determined if the capacities of the physical device is known. For example, if the number of the available bits per track is known, we can determine how many tracks we need to store the collection of data.

Assuming for coding of x objects, we need $\log_2 x$ each keyword is coded with $\log_2 N$ bits. Within a node there are three portions, namely a code for a keyword a code for a storage device address which is a pointer to its son, and a code for list length. Let us assume each node have M bits, therefore

$$S_D = MD = \frac{m}{m - 1} N M$$

Let us define:

L = average list length.

R = number of records.

K = average number of keywords per record.

N = number of keywords.

It is clear that $L = \frac{K \times R}{N}$ and $S_I = NL \log_2 R$. Eventhough we can assume that the collection of lists contains as much information as the data file, if there is an inverted list for every keyword and every record is defined by the subset of keywords, the reconstruction of a record from the collection of inverted lists is very inefficient. Therefore we still have to store the data file for the ease of retrieval. We can calculate S_A by

$$S_A = R \times K \times \log_2 N$$

Now, the total storage requirement for an inverted file is

$$S_2 = S_D + S_I + S_A$$

$$= \frac{m}{m-1} N M + N L \log_2 R + R \times K \times \log_2 N$$

Consider the previous example, N=1000, $L=2 \times 10^4$, $R=10^6$, K=20, u=10 and let m=3 and M=100, we have

$$S_2 = 3/2 \times 10^3 \times 100 + 10^3 \times 2 \times 10^4 \times 20 + 10^6 \times 20 \times 10$$

$$= 1.5 \times 10^5 + 4 \times 10^8 + 2 \times 10^8$$

$$= 6 \times 10^8 \text{ bits}$$

For the proposed file structure under the same assumption, we have

$$S = 100 \frac{10 \times 10^6}{10 - 1} = 1.1 \times 10^8 \text{ bits.}$$

From these crude estimates, we can conclude that

(1) The storage requirement and the query processing time, especially for large file and complex queries, the proposed file structure has the advantage over the inverted file.

(2) The CPU processing time for a query in the proposed file structure depends on a very sensitive parameter r, for $r \geq \frac{1}{2}$, the CPU processing time may be larger than that of the inverted file.

(3) The query processing time of the proposed file structure is not directly proportional to the number of keywords in a query and the list lengths. Thus, we conjecture that the proposed file structure has an uniform processing time.

The file maintanance problem for the proposed file structure is similar to the problem of retrieval. For example, to add a record to the file, a set of Boolean functions has to be evaluated to determine what part of the tree this record will belong to, then the record is broken up into fragments which correspond to nodes in the tree. But for the inverted file, to update the file is to update the index decoder, the collection of inverted lists and the data file. By this simple reasoning, we believe that the proposed file structure will also has an advantage over the inverted file.

V. CONCLUSION

We have defined a file as a collection of functions mapping from a set of records to some value sets. A homogeneous subfile is defined as a subset of a space of the product of some value spaces, i.e. a relation. In general, a file consists of a collection of homogeneous subfiles. Thus, our definition of a file is equivalent to Codd's relational model of a file which is defined as a collection of normalized relations. [4]

In this thesis, we have introduced a new file structure which is based on the concept of atoms in a Boolean algebra. Starting with the assumption that the allowable queries are Boolean functions of keywords, we can view the process of answering a query as evaluating a homomorphism which maps a Boolean algebra of queries into a Boolean algebra of the file. From this point of view, a file structure based on the atoms of the Boolean algebra of the file arises naturally. For implementation purposes, we have found it advantageous to represent our file structure by a nested sequence of partitions of the file. With this representation, evaluating a query becomes the problem of successive approximations of a

Boolean function. With the representation of the file as a nested sequence of partitions, trees and binary trees become natural choices for the data structure. A program incorporating these ideas has been written and run on some randomly generated data. The purpose of writing this program was to confirm feasibility. As written, it is neither a general data management package nor an adequate test program for alternative strategies. Nevertheless, we feel that the experience that we have obtained in running this program has simply confirmed the feasibility of the basic idea. Our experience from running this program and from a simple analysis have also demonstrated that our proposed structure should compare extremely favorable against competing structures, e.g. inverted file and multilist, with respect to both storage requirement and retrieval time.

We conclude that the Boolean algebra representation is a natural basic model for a file. Once we have a basic model, more complex structures can be constructed for a file organization. For example, by defining a set of operators on a set of homogeneous subfiles, we will have a relational model of a file [4, 5], since a relational algebra is a Boolean algebra with a set of additional operators.[34] The tree

representation of data is not new, but the concept of indexing by Boolean functions is believed to be novel. The sample file is implemented in the core memory of a CDC computer. The techniques we use can be applied immediately to a computer system with virtual memory and associated programming facilities. [37,38]

For future work, we plan to study other aspects of Boolean algebras applicable to our file structure. For example, we will study the possibility of finding a reasonable way for selecting the sets of generators of those nested Boolean algebras, such that the tree representation will have a desired form. The separability of a file is also worth studying. If a file can be represented by a linearly separable Boolean function, then answering a query can be viewed as finding the intersection of the query with the separating hyperplane [31] of the Boolean function representing the file. The problem of constructing more complex structure onto the file, say relational algebras, is also an interesting area to be investigated.

REFERENCES

1. CODASYL Language Structure Group (1962). "An information algebra," CACM Vol. 5, (April 1962), pp190-204.

2. McGee, William C. (1962). "The property classification method of file design and processing," CACM Vol. 5, (1962), pp45-458.

3. McGee, William C. (1968). "File structures for generalized data management," IFIP Congress 68, 1968, pp68-73.

4. Codd, E.F. (1970). "A relational model of data for large stored data banks," CACM Vol.13, No.6, (June 1970), pp377-387.

5. Codd, E.F. (1971). "A data base sublanguage founded on the relational calculus," IBM Research, RJ893, (July 1971).

6. Hsiao, David and Harary, Frank (1970). "A formal system for information retrieval from files," CACM Vol.13, No.2, (February 1970), pp67-73.

7. Strand, Alois J. (1971). "The relational approach to the management of data bases," IFIP Congress, 1971, pp591-594.

8. Goldstein, Robert C. and Strand, Alois J. (1970). "The MacAims data management system," ACM SICFIDET Workshop on Data Description And Access, 1970, pp201-229.

9. Fry, J.P. et al (1969). "Data management systems survey," The Mitre Corp., Report AD684707, (January 1969).

10. Collmeyer, A.J. (1970). "File organization techniques," IEEE Computer Group News, (March-April 1970), pp3-11.

11. Lefkovitz, David (1969). File Structure For On-line Systems, Spartan Books, New York.

12. Senko, M.E. (1969). "File organization and management information systems," Annual Review of Information Science and Technology, Vol.4, Encyclopedia Britanica Inc., Chicago, pp111-143.

13. Senko, M.E. et al 91967). Formatted File Organization Techniques, Final Report, IBM Research, (May 1967).

14. Chaplin, Ned (1969). "Common file organization techniques compared," AFIPS Conference Proceedings, 1969, FJCC, Vol.35, pp413-422.

15. Dzubak, B.J. and Warburton, C.R. (1965). "The organization of structured files," CACM Vol.8, No.7, (July 1965), pp446-452.

16. Angell, Thomas and Randell, Theron M. (1969). "Generalized data management systems," IEEE Computer Group News, Vol.2, No.12, (November 1969), pp5-12.

17. Blier, Robert E. and Vorhaus, Alfred H. (1968). "File organization in the SDC time shared data management(TDMS)," IFIP Congress Proceedings, 1968, pp92-97.

18. Bryant, J.H. and Semple, Parlan, Jr. (1966). "GIS and file management," Proceedings ACM National Meeting, 1966, pp97-107.

19. Lombardi, L. (1962). "Mathematical structure of non-arithmatic data processing," JACM Vol.9, 1962, pp136-159.

20. Namin, Paul (1968). "Algebra of management information," IFIP Congress Proceedings, 1968, pp63-67.

21. Sussenguth, Edward H, Jr.(1963). "Use of tree structures for processing files," CACM Vol.6, No. 5, (May 1963), pp272-279.

22. Wong, E. and Chiang, T.C. (1971). "Canonical structure in attribute based file organization," CACM Vol.14, No.9, (September 1971), pp 93-97.

23. Petterson, W.W.(1957). "Addressing for random access storage," IBM Journal,(April 1957), pp130-146.

24. Morris, Robert (1968). "Scatter techniques," CACM Vol.11, No.1, (January 1968), pp38-44.

25. Morris, Robert (1969). "Some theorem on sorting," SIAM Appl. Math., Vol.17, No.1, (January 1969), pp1-6.

26. Buchholz, Werner (1963). "File organization and addressing," IBM Systems Journal, (June 1963), pp86-111.

27. Wong, E. (1964). "A linear search problem," SIAM Review, Vol.6, No.2, (April 1964), pp168-174.

28. Knuth, D.(1971). "Optimal binary search trees," Acta Informatica, Vol.1, 1971, pp14-25.

29. Tausworthe, Robert C. (1965). "Random number generated by linear recurrence modulo two," Mathematics of Computation, Vol.19, No.90, (April 1965), pp201-209.

30. Holt, A.W., et al (1969). A Handbook On File Structuring, Final Report, Applied Data Research Inc., Princeton, N.J., September 1969.

31. Hu, S.T. (1968). Mathematical Theory of Switching Circuits and Automata, University of California Press, Los Angeles, California.

32. Halmos, P.R. (1960). Naive Set Theory, Van Nostrand, First Edition, New York.

33. Halmos, P.R. (1967). Lectures on Boolean Algebras, Van Nostrand, New York.

34. Sikorski, Roman (1969). Boolean Algebras, Springer-Verlag, New York.

35. Kleene, S.C. (1968). Mathematical Logic, John Wiley and Sons Inc., New York.

36. Knuth, D.E.(1968). The Art of Computer Programming, Addison-Wesley Co., Palo Alto, California.

37. Cobato, F.J. and Vyssotsky, V.A. (1965). "Introduction and overview of the Multics system," FJCC Proceedings, 1965, pp185-196.

38. Denning, P.J.(1970). "Virtual Memory," Computing Surveys, Vol.12, No.3, (September 1970), pp153-189.

APPENDIX

This appendix contains listings of programs which are written for the implementation of the sample file and for the process of Boolean queries. Results from several runs of the programs are also included in this appendix. A set of queries is tested on the programs. Processing times are recorded by making a call to a library subroutine , SECOND.

The listings of programs include statements for the following procedures:

(1). TREE
(2). CONVERT
(3). DETREE
(4). RETRIEV
(5). ADDBIT
(6). RETRI
(7). MATCH
(8). MASH

The listings of outputs from the programs include :

(1). The set of randomly generated records
(2). The tree representation
(3). A set of queries with various number of keywords involved
(4). Sets of records pertaining to a query
(5). Processing times for queries

Due to the bulk of the computer print-out, only a part of the set of records and a part of the tree representation are listed.

```
 RUN FORTRAN COMPILER VERSION 2.3 B.2

                      PROGRAM TREE (INPUT,OUTPUT)
000003                INTEGER B(2000),TR(5000),PT
000003                COMMON K
000003          1     FORMAT (2020)
000003          15    FORMAT (/2X,*NUMBER OF NODES = *,I6//2X,*SIZE OF EACH NODEIS
                     1 ONE COMPUTER WORD*)
000003          14    FORMAT(2X,020)
000003          12    FORMAT (/2X,*HERE IS THE DECODING TREE *//(2X,020))
000003          111   FORMAT (I4)
000003          17    FORMAT (//2X,*NO. OF KEYWORDS IN THE QUERY*,//(4X,I10))
000003          222   FORMAT (2I4)
000003          2     FORMAT(2X,*NO. OF RECORDS  NO. OF BITS PER CODE*//(2X,2I4))
000003          3     FORMAT (2X,*THE SET OF RECORDS*//(2X,2020))
000003                READ  222, N,K
000013                READ 1,(B(I), I=1,N)
000026                PRINT 2, N,K
000036                PRINT 3,(B(I),I=1,N)
000051                DO  100 I=1,5000
000053                TR(I)=0
000054          100   CONTINUE
000056                PT=LOCF(TR)
000060                DO 9 I=1,N,2
000061                M=0
000062                CALL CONVERT (B,I,M)
000065                PRINT 14,M
000073                NL=1
000074                J=0
000075                I1=I
000077          4     CONTINUE
000077                I3=(I+1)/2
000102                CALL DETREE (M,NL,TR,PT,I3)
000105          11    M=B(I1)
000107                IF (J.EQ.6) GO TO 10
000111                JJ=J*K
000113                IF (JJ.EQ.0) GO TO 13
000114                CALL LSHIFT (M,JJ)
000116          13    M=777400000000000000B.AND.M
000120          5     NL=NL+1
000122                J=J+1
000123                IF (NL.GT.K+1) GO TO 9
000127                IF (M.EQ.0) GO TO 11
000130                GO TO 4
000130          10    I1=I1+1
000132                J=0
000133                GO TO 11
000133          9     CONTINUE
000136                PRINT 12,(TR(I), I=1,5000)
000147                NODE=0
000150                DO 16 I=1,5000
000152                IF (TR(I).EQ.0) GO TO 16
000153                NODE=NODE+1
000155          16    CONTINUE
000157                PRINT 15, NODE
```

```
RUN FORTRAN COMPILER VERSION 2.3 B.2

000165           IK1=1
000166    444    CONTINUE
000166           READ  111,NKK
000174           PRINT 17,NKK
000202           CALL  RETRIEV (TR,KE,NKK)
000205           IK1=IK1+1
000207           IF (IK1.LE.10) GO TO 444
000211           END
```

RUN FORTRAN COMPILER VERSION 2.3 B.2

```
                SUBROUTINE CONVERT(A,KK,MT)
000006          INTEGER  A(2000)
000006          COMMON K
000006          MT=0
000006          JA=0
000007          K2=KK+1
000011          DO 6 I=KK,K2
000012          J=0
000013     5    JJ=J*K
000015          IA=A(I)
000017          IF (JJ.EQ.0) GO TO 11
000021          CALL  LSHIFT (IA,JJ)
000023    11    IT=777400000000000000000B.AND.IA
000025          IF(IT.EQ.0) GO TO 3
000030          GO TO 4
000031    3     J=J+1
000033          JA=JA+1
000034          IF(J.EQ.6) GO TO 6
000036          IF (JA.EQ.10) GO TO 6
000040          GO TO 5
000040    4     IF(JA.EQ.0) GO TO 12
000041          IT=400000000000000000000B
000043          CALL RSHIFT(IT,JA)
000045          GO TO 13
000050    12    IT=400000000000000000000B
000052    13    MT=MT.OR.IT
000053           JA=JA+1
000055              IF(J.EQ.5) GO TO 6
000057          IF(JA.EQ.10) GO TO 6
000061          J=J+1
000062          GO TO 5
000063    6     CONTINUE
000066          RETURN
000066          END
```

```
          IDENT  DETREE
PROGRAM LENGTH

BLOCKS

PROGRAM*   LOCAL

ENTRY POINTS

   000000 DETREE

          ENTRY  DETREE
 DETREE   BSS  1
          SA1  B3
 L2       NZ  X1,L4
          SA2  B1
          BX1  X2
          SB6  A1
          NE  B3,B6,L3
          MX3  33
          MX2  12
          BX2  X3-X2
          BX1  X1+X2
          SA4  B5
          BX4  -X4
          LX4   6
          MX3  54
          MX2  33
          BX2  X3-X2
          BX2  X2*X4
          BX1  X1+X2
          SA2  B2
          BX1  X1+X2
           SA4  B4
          SX6  X4+1
          SA6  B4
          BX6  X1
          SA6  A1
          EQ B0,B0,DETREE
 L3       SA2  B2
          BX1  X1+X2
          SA4  RN
          LX4  6
          MX3  54
          MX2  33
          BX2  X3-X2
          BX2  X2*X4
          BX1  X1+X2
          SA4  B4
          SX4  X4-1
          BX4  -X4
          LX4  27
          MX3  33
```

```
          MX2   12
          BX2   X3-X2
          BX2   X2*X4
          BX1   X1+X2
          SA4   B4
          SX6   X4+1
          SA6   B4
          BX6   X1
          SA6   A1
          EQ    B0,B0,DETREE
L4        SA4   B2
          MX3   60
          MX2   54
          BX2   X3-X2
          BX2   X1*X2
          BX4   X4-X2
*         ZR    X4,L8
          MX3   54
          MX2   33
          BX2   X3-X2
          BX2   X1*X2
          LX2   33
          PL    X2,L7
          AX2   39
          BX6   X2
          SA6   RN
          SA4   B4
          LX4   6
          MX3   54
          MX2   33
          BX2   X3-X2
          BX4   X2*X4
          BX2   -X2
          BX1   X2*X1
          BX6   X1+X4
          SA6   A1
          SA1   B4
          SA1   X1
          EQ B0,B0,L2
L7        SA2   NEXT
*         NZ    X2,L13
          SA1   B4
           SA1   X1-1
          EQ B0,B0,L2
L8        SA2   B1
          MX3   12
          BX3   X1*X3
          BX2   X3-X2
          NZ    X2,L10
          MX3   54
          MX2   33
          BX2   X3-X2
          BX2   X1*X2
```

```
                LX2   33
                PL    X2,L14
                EQ    B0,B0,DETREE
L10             MX3   33
                MX2   12
                BX2   X3-X2
                BX3   -X2
                BX2   X1*X2
                BX3   X1*X3
                BX6   X2
                AX6   27
                LX2   12
                PL    X2,L12
                SA6   TEMP
                SA4   B4
                LX4   27
                BX6   X3+X4
                SA6   A1
                SA1   B4
                SA1   X1
                SA2   B1
                BX1   X1+X2
                SA2   B2
                BX1   X1+X2
                SA2   TEMP
                LX2   27
                MX3   33
                MX4   12
                BX4   X3-X4
                BX2   X2*X4
                BX1   X1+X2
                SA2   B5
                BX2   -X2
                LX2   6
                MX3   54
                MX4   33
                BX4   X3-X4
                BX2   X2*X4
                BX1   X1+X2
                SA2   B4
                SX6   X2
                SA6   NEXT
                SX6   X2+1
                SA6   B4
                BX6   X1
                SA6   A1
                EQ    B0,B0,DETREE
L12             MX3   33
                MX2   15
                BX2   X3-X2
                BX2   X1*X2
                AX2   27
                SA1   X2
```

```
*               EQ   B0,B0,L8
    L13         SA1  X2
                SX6  000000B
                SA6  NEXT
*               EQ   B0,B0,L2
    L14         MX3  54
                MX2  36
                BX2  X3-X2
                BX2  X1*X2
                AX2  6
                BX6  X2
*               SA6  NEXT
                EQ   B0,B0,DETREE
    NEXT        BSSZ 1
    TEMP        BSSZ 1
    RN          BSSZ 1
                END

        166 STATEMENTS          13  SYMBOLS
```

```
          IDENT  RSHIFT
PROGRAM LENGTH

BLOCKS

PROGRAM#   LOCAL

ENTRY POINTS

   000000 RSHIFT

          ENTRY  RSHIFT
RSHIFT    BSS  1
          SA1  B1
          SA2 B2
          SB3  X2
          AX3  B3,X1
          BX6  X3-X1
          SA6  B1
          SB3  X2-1
          EQ  B3,B0,RSHIFT
          AX4  B3,X1
          BX6  X3-X4
          SA6  B1
          JP  RSHIFT
          END

       16 STATEMENTS          1  SYMBOLS

          IDENT   LSHIFT
PROGRAM LENGTH

BLOCKS

PROGRAM#   LOCAL

ENTRY POINTS

   000000 LSHIFT

          ENTRY    LSHIFT
 LSHIFT    BSS   1
          SA1  B1
          SA2  B2
          SB3  X2
          LX3  B3,X1
          BX6  X3
         SA6  B1
          JP  LSHIFT
          END

       11 STATEMENTS          1  SYMBOLS
```

ORTRAN COMPILER VERSION 2.3 B.2 28 AU

```
      SUBROUTINE  RETRIEV (TR1,KEY,NK)
      DIMENSION KEY(NK)
      DIMENSION IQUERY(4)
5     FORMAT (2X,*RECORDS SATISFYING THE QUERY*)
300   FORMAT (2X,*CPU TIME FOR ANSWERING THIS QUERY. IN SEC. *//(F10.7))
200   FORMAT (2X,*END OF FILE SEARCH*)
4     FORMAT (2X,*QUERY*)
      PRINT 4
      CALL SECOND (T1)
      ENCODE (10, 2, FM) NK
2     FORMAT (2H( ,I4, 3HI4))
      READ FM, (KEY(I), I=1,NK)
      PRINT FM, (KEY(I), I=1,NK)
      M=777777777777777777B
      NB=0
          NBIT=0
      K1=1
      K2=K1+9
      N=100
14    CONTINUE
      DO 12 I=1,NK
      IF (KEY(I).GT.0) GO TO 3
      GO TO 12
3     CONTINUE
      KEYC=KEY(I)
33    CONTINUE
      IF (KEYC.LE.K2.AND.KEYC.GE.K1) GO TO 7
6     K1=K1+10
      IF (K1.GT.100) GO TO 11
      K2=K1+9
      GO TO 33
7     NBIT=(KEYC-1)/10 +1
      IF (NB.EQ.NBIT) GO TO 12
      NB=NBIT
8     CALL ADDBIT (M,NBIT)
12    CONTINUE
11    CONTINUE
      DO 100 I=1,4
      IQUERY(I)=777777777777777777B
100   CONTINUE
      DO 108  I=1,NK
101   IF(KEY(I).GT.0) GO TO 102
      GO TO 103
102    CONTINUE
      IF(KEY(I).GT.60) GO TO 104
      CALL ADDBIT(IQUERY(1),KEY(I))
      GO TO 108
104    KCON=KEY(I)-60
      CALL ADDBIT(IQUERY(2),KCON)
      GO TO 108
103   CONTINUE
      IF(-KEY(I).GT.60) GO TO 105
      KCON=-KEY(I)
```

```
 RUN: FORTRAN COMPILER VERSION 2.3 B.2

000211           CALL ADDBIT(IQUERY(3),KCON)
000214           GO TO 108
000217    105     KCON=-KEY(I)-60
000222           CALL ADDBIT (IQUERY (4),KCON)
000225    108    CONTINUE
000232           PRINT 5
000235           CALL RETRI  (TR1,IQUERY,M)
000241           CALL SECOND(T2)
000243           T3=T2-T1
000245           PRINT 300,T3
000253           PRINT  200
000257           RETURN
000260           END
```

```
          IDENT ADDBIT
PROGRAM LENGTH

BLOCKS

PROGRAM*   LOCAL

ENTRY POINTS

   000000 ADDBIT

          ENTRY  ADDBIT
 ADDBIT  BSS   1
          MX1   1
          SA2   B1
          SA3   B2
          SB3   X3-1
*         EQ    B0,B3,L1
       AX4 B3,X1
          SB3   B3-1
          AX3   B3,X1
          BX1   X4-X3
 L2       BX6   X2-X1
          SA6   B1
*         EQ    B0,B0,ADDBIT
 L1       JP    L2
          END

       17 STATEMENTS          3  SYMBOLS
```

```
          IDENT  RETRI
PROGRAM LENGTH

BLOCKS

PROGRAM*   LOCAL

ENTRY POINTS

   000000 RETRI

EXTERNAL SYMBOLS

PRIN       MASH       MATCH

           ENTRY  RETRI
           EXT  PRIN
           EXT  MASH
           EXT  MATCH
 RETRI     BSS  1
 IXX*X     OPDEF  I,J,K
           PX,J  B0,X,J
           PX,K  B0,X,K
           DX,I  X,J*X,K
           UX,I  B7,X,I
           UX,K  B7,X,K
           UX,J  B7,X,J
           ENDM
 SAVE      MACRO  I,J,K
           SX6  B,I
           SA6  S1
           SX6  B,J
           SA6  S2
           SX6  B,K
           SA6  S3
           ENDM
 SETT  MACRO I,J,K
           SA1  S1
           SB,I X1
           SA1  S2
           SB,J  X1
           SA1  S3
           SB,K  X1
           ENDM
           SX6  10
           SA6  K
           MX6  60
           SA6  ONE
           SX6  0000008
           SA6  J
           SX6  B1
           SA6  PTR
 L2         SA2  PTR
```

```
                SA2   X2
                MX3   12
                BX2   X2*X3
                SA3   B3
                BX2   X2+X3
                SA1   ONE
                BX1   X1-X2
                ZR    X1,L5
        L3      SA2   PTR
                SA2   X2
                MX3   33
                MX4   12
                BX3   X3-X4
                BX1   X2*X3
                LX1   12
                AX1   39
                SA2   ONE
                BX2   X2-X1
                NZ    X2,L4
*               JP    RETRI
        L4      BX6   X1
                SA6   PTR
                JP    L2
        L5      MX3   54
                MX4   33
                BX3   X3-X4
                SA2   PTR
                SA2   X2
                BX6   X2*X3
                AX6   6
                SA6   PTR
        L6      SA1         J
*               SA2   K
                IX6   X1*X2
*               SA6   JJ
                SX2   2
                IX6   X1+X2
                SA6   NLQ
                SAVE    1,2,3
*               SB1   JJ
                SB3   MM
X               RJ    MASH
                SETT  1,2,3
                SA1   MM
                SA2   A1+1
                AX1   48
                AX2   48
                BX1   -X1
                BX2   -X2
                BX1   X1+X2
                ZR    X1,L16
        L9      SAVE    1,2,3
*               SB1   PTR
```

```
                SB2   MM
      +         SB3   NLQ
                SB4   MN
      X         RJ    MATCH
                SETT  1,2,3
                SA1   MN
      +         SA2   ONE
                BX2   X1-X2
                ZR    X2,L16
                ZR    X1,L19
          L11   SA1   PTR
                SA1   X1
                LX1   12
                NG    X1,L13
                AX1   39
                MX2   60
                MX3   39
                BX2   X2-X3
                BX1   X1*X2
                BX6   X1
                SA6   PTR
      +         JP    L9
          L13   AX1   39
                BX6   -X1
      +         SA6   PTR
                SA1   X6
                MX2   60
                MX3   54
                BX2   X2-X3
                BX2   X1*X2
                SX6   X2-2
                SA6   J
                SX6   X6+1
                ZR    X6,L21
      +         SA1   PTR
                SA1   X1
                LX1   12
      +         NG    X1,L13
                AX1   39
                MX2   60
                MX3   39
                BX2   X2-X3
                BX6   X1*X2
                SA6   PTR
                JP    L6
          L21   SX6   000000B
      +         SA6   J
                JP    L3
          L16   SA1   PTR
                SA1   X1
                MX2   54
                MX3   33
                BX2   X2-X3
```

```
                BX2   X1*X2
                LX2   33
                PL    X2,L18
                AX2   39
                BX6   -X2
                SA6   REC
                SAVE      1,2,3
*               SB1   REC
                RJ    PRIN
                SETT  1,2,3
                JP    L11
        L18     AX2   39
                BX6   X2
*               SA6   PTR
*       L19     SA1   J
                SX6   X1+1
                SA6   J
*               JP    L6
        K       BSSZ  1
        JJ      BSSZ  1
        J       BSSZ  1
        VLQ         BSSZ  1
        PTR         BSSZ  1
        S1          BSSZ  1
        S2      BSSZ  1
        S3      BSSZ  1
        REC     BSSZ  1
        ONE     BSSZ  1
        MM      BSSZ  2
        MN      BSSZ  2
                END

     243 STATEMENTS          28  SYMBOLS
```

```
                    SUBROUTINE  PRIN(C)
000003              PRINT 1, C
000011    1         FORMAT (2X,*RECORD*,2X,I10)
000011              RETURN
000012                  END
```

```
          IDENT  MATCH
PROGRAM LENGTH

BLOCKS

PROGRAM*   LOCAL

ENTRY POINTS

   000000 MATCH

          ENTRY  MATCH
 MATCH    BSS  1
          MX6  60
          SA6  MONE
          SA1  B1
          SA1  X1
          SA2  B3
          MX3  60
          MX4  54
          BX3  X3-X4
          BX3  X1*X3
          BX2  X3-X2
          ZR  X2,L9
          SX6  B0
          SA6  B4
          JP  MATCH
 L9       SA2  B2
          SA3  MONE
          BX3  X2-X3
          ZR  X3,L6
          MX3  12
          BX3  X1*X3
          SA2  B2
          BX3  X2+X3
          SA2  MONE
          BX3  X2-X3
          ZR  X3,L6
          SX6  1
          SA6  B4
          JP  MATCH
 L6       SA2  B2+1
          MX3  12
          BX3  X1*X3
          BX3  -X3
          BX3  X2+X3
          BX3  -X3
          ZR  X3,L10
          SX6  1
          SA6  B4
          JP  MATCH
 L10      MX6  60
          SA6  B4
          JP  MATCH
 MONE     BSS  1
          END

       46 STATEMENTS          5 SYMBOLS
```

```
          IDENT   MASH
PROGRAM LENGTH                                          120

BLOCKS

PROGRAM*    LOCAL

ENTRY POINTS

   000000 MASH

          ENTRY   MASH
MASH      BSS   1
TEMP      MACRO   P2,K
          SA1   B2+K
          BX6   X1
          SA6   P2
          SA1   A1+2
          BX6   X1
          SA6   A6+1
          ENDM
MASK      MACRO   K
          SA1   TEMP
          SA1   A1+K
          MX2   10
          BX6   X1*X2
          MX1   60
          BX1   X1-X2
          BX6   X1+X6
          SA2   B3
          SA2   A2+K
          SA6   A2
          ENDM
          SA1   B1
          ZR    X1,L12
          SX1   X1-60
          NG    X1,L3
          JP    L8
L12       TEMP    TEMP,0
          JP    L7
L3        TEMP    TEMP,0
L4        SA1   TEMP
      SA2   B1
          SB4   X2
          LX6   B4,X1
          SA6   TEMP
          SA1   A1+1
          LX6   B4,X1
          SA6   A1
L7        MASK    0
          MASK    1
          JP    MASH
L8        SX6   X1
          ZR    X6,L11
          SA6   B1
L10       TEMP    TEMP,1
          JP    L4
L11       TEMP    TEMP,1
          JP    L7
TEMP      BSS   2
          END
```

NO. OF RECORDS NO. OF BITS PER CODE

998 10

THE SET OF RECORDS

744114046023011403072434736357167473610 3
636751364572275137012121306543261530650 1
606067433615706754772023553665732755242 5
176244722351164463630521434616307143412 1
031736557267533640617671122451224512375 7
522706143061430605334616757367573675746 7
357224512245122453556452143061430614264 5
234604102041020415436136037017407603644 1
206601300540260123411755770774376177164 1
053325152465232512377465146463231514661 3
535704342161070435713131036417207503645 1
671076437217507654312273523651724752336 7
162646123051424606557216357167473635623 3
260613305542661321671575710744362171172 3
045327153465632712617431150464232115062 1
241073035416607315611404266133055426732 1
737762371174476231172651216507243521670 5
623060430214106042752167575676737357562 5
605532455226513252152030662331154466317 3
601355766773375567272005332555266533351 5
143026213105442620157367621710744362000 5
512256127053425617754715456627313545725 3
427334556267133446314152643321550664221 1
373223711744762376071265612705342561257 5
715457027413605703252733451624712345166 3
626534656327153465575623135455627313544 5
456271334556267124353456271134456227141 3
643215106443221504075411730754366173140 3
337051024412205113771124306143061430712 5
706367573675736747155022553265532655350 5
252525252525252525256314630314146063031 4
210756767373575676731702455226513245522 7
727634516247123451635151642721350564305 7
473323751764772370333511164472235116501 1
130705542661330547717101032415206503352 7
507701140460230104634752771374576277142 1
424552665332555264174143315546663331437 3
404267333555666736513742227113445622654 7
124161470634316143350637205502641320411 3
756517447623711740712454724352165072402 7
301633115446623313036772111044422211067 7
552570674336157075534463275536657327533 1
435662531254526246674131562671334554222 3
406732155066433216174022661330554266120 5
401555666733355566034007333555666733220 1
377511044422211101771253071434616307005 3
714502641320550277475043015406603301524 3
474277337557667731413536251124452225073 7
646614706343161462652356043021410604215 5
164574076037017417330543241520650324051 1
327234516247123447376626135056427213426 5

```
6127561670734356124357164572275136457141
6415264532255126405323706223111444623747
4470144062031014420134277357567673734177
6734762371174476320555102561270534256603
5247073435616707222331410264132055027617
0063321550664332133177551164472235116667
0376623311544662267701255606703341561553
1415730754366173150504072675336557257303
7476277137457627712524261530654326152715
3765112445222511362565207063431614707215
4051542661330554270540333375576677336275
4203355566673335532141766453225512645117
5212531254526253062146054666333155467561
0240072035016407372376377517647723751261
1054324152065032423307441140460230114167
5235604702341160536731645742761370574655
0352625312545262445176462151064432215431
1505700740360170061304772775376577277571
7126307143461630764127161030414206103237
2742115046423211456562420730354166073455
1156571274536257024770463271534656327635
5720234116047023507532636137057427612753
2225352565272535331361606463231514647107
3310502241120450274366734776377177477537
6746115046423211515155357047423611705047
5176577277537657771130523251524652325271
1610202101040420235172076017007403600131
6645214506243121551522346043021410604473
5460274136057027517333576257127453624727
1713103441620710340305067013405602701377
6353517647723751741321313045422611304407
7515164472235116544524707223511644722435
1171644722351164566307272355166473234557
5632463231514646257532123571674736356325
2774776377177477621162535256527253525607
1003001400600300125370010004002001000631
5017003401600700200530050004002001001775
1073011404602301002770250064032015006015
5327053425612705211331150304142061030071
3403733755766773244113765112445222510341
5400266133055426664544001540660330155543
3411156467233515746764047323551664733355
5445726753365572775744372635316547252533
3676211104442221012165521710744362170061
4503255526653325442543026333155466632363
6112031014406203050157057667733755767477
5561712745362571215544572715346563271733
4346627313545662651136422151064432214307
6542173075436617210323360510244122051701
6000666333155466647357742221110444223351
4524160070034016013134637217507643721711
6421260530254126127123606203101440621151
1654216107043421704772336057027413604743
5166575276537257553747214526253125453313
4263347563671734710736211344562271135075
1217065432615306475371710224112045022247
```

HERE IS THE DECODING TREE

77747777777000432401
74400004336000432502

2300777345300043260 3
2300777345200043270 4
2300777345100043300 5
2300777345000043310 6
1434777344700043320 7
2434777344600043331 0
1674777344500043341 1
1674777344400043351 2
1674777344377777761 3
6364000435000043370 2
7224777344100043400 3
7224777344000043410 4
7224777343700043420 5
7224777343600043430 6
7404777343500043440 7
2120777343400043451 0
2614777343300043461 1
2614777343200043471 2
2614777343177777751 3
6060000436200043510 2
1570777342700043520 3
1570777342600043530 4
1570777342500043540 5
1570777342400043550 6
6374777342300043560 7
2020777342200043571 0
7324777342100043601 1
7324777342000043611 2
7324777341777777741 3
1760000437400043630 2
5114777341500043640 3
5114777341400043650 4
5114777341300043660 5
5114777341200043670 6
1714777341100043700 7
0520777341000043711 0
3070777340700043721 1
3070777340600043731 2
3070777340577777731 3
0314000440600043750 2
6750777340300043760 3
6750777340200043770 4
6750777340100044000 5
6750777340000044010 6
0304777337700044020 7
7670777337600044031 0
2244777337500044041 1
2244777337400044051 2
2244777337377777721 3
5224000442000044070 2
6140777337100044100 3
6140777337000044110 4
6140777336700044120 5
6140777336600044130 6
2554777336500044140 7
4614777336400044151 0

5734777336300044161 1
5734777336200044171 2
5734777336177777711 3
3570000443200044210 2
4510777335700044220 3
4510777335600044230 4
4510777335500044240 5
4510777335400044250 6
5664777335300044260 7
6450777335200044271 0
4304777335100044301 1
4304777335000044311 2
4304777334777777701 3
2344000444400044330 2
4100777334500044340 3
4100777334400044350 4
4100777334300044360 5
4100777334200044370 6
6614777334100044400 7
6134777334000044411 0
4074777333700044421 1
4074777333600044431 2
4074777333577777671 3
2064000445600044450 2
4024777333300044460 3
4024777333200044470 4
4024777333100044500 5
4024777333000044510 6
1604777332700044520 7
1754777332600044531 0
3760777332500044541 1
3760777332400044551 2
3760777332377777661 3
0530000447000044570 2
6520777332100044600 3
6520777332000044610 4
6520777331700044620 5
6520777331600044630 6
5174777331500044640 7
7464777331400044651 0
2314777331300044661 1
2314777331200044671 2
2314777331177777651 3
5354000450200044710 2
6104777330700044720 3
6104777330600044730 4
6104777330500044740 5
6104777330400044750 6
6744777330300044760 7
3130777330200044771 0
2074777330100045001 1
2074777330000045011 2
2074777327777777641 3
6710000451400045030 2
1750777327500045040 3
1750777327400045050 4

2370777264300051360 5
2370777264200051370 6
0344777264100051400 7
2454777264000051411 0
1650777263700051421 1
1650777263600051431 2
1650777263577777271 3
3014000515600051450 2
4660777263300051460 3
4660777263200051470 4
4660777263100051500 5
4660777263000051510 6
5414777262700051520 7
6770777262600051531 0
4220777262500051541 1
4220777262400051551 2
4220777262377777261 3
5524000517000051570 2
3614777262100051600 3
3614777262000051610 4
3614777261700051620 5
3614777261600051630 6
6654777261500051640 7
4460777261400051651 0
6570777261300051661 1
6570777261200051671 2
6570777261177777251 3
4354000520200051710 2
5450777260700051720 3
5450777260600051730 4
5450777260500051740 5
5450777260400051750 6
3334777260300051760 7
4130777260200051771 0
3344777260100052001 1
3344777260000052011 2
3344777257777777241 3
4064000521400052030 2
6640777257500052040 3
6640777257400052050 4
6640777257300052060 5
6640777257200052070 6
7074777257100052100 7
4020777257000052111 0
5540777256700052121 1
5540777256600052131 2
5540777256577777231 3
4014000522600052150 2
3334777256300052160 3
3334777256200052170 4
3334777256100052200 5
3334777256000052210 6
3014777255700052220 7
4004777255600052231 0
6664777255500052241 1
6664777255400052251 2

243477732037777756 13
623000046100004577 02
141077732010004600 03
141077732000004601 04
141077731770004602 05
141077731760004603 06
136477731750004604 07
216477731740004605 10
737077731730004606 11
737077731720004607 12
737077731717777755 13
605400046220004611 02
265077731670004612 03
265077731660004613 04
265077731650004614 05
265077731640004615 06
506477731630004616 07
203077731620004617 10
154477731610004620 11
154477731600004621 12
154477731577777754 13
601000046340004623 02
733477731550004624 03
733477731540004625 04
733477731530004626 05
733477731520004627 06
353477731510004630 07
200477731500004631 10
266477731470004632 11
266477731460004633 12
266477731457777753 13
143000046460004635 02
054477731430004636 03
054477731420004637 04
054477731410004640 05
054477731400004641 06
006477731370004642 07
736477731360004643 10
744077731350004644 11
744077731340004645 12
744077731337777752 13
512000046600004647 02
534077731310004650 03
534077731300004651 04
534077731270004652 05
534077731260004653 06
776477731250004654 07
471477731240004655 10
313477731230004656 11
313477731220004657 12
313477731217777751 13
427000046720004661 02
671077731170004662 03
671077731160004663 04
671077731150004664 05
671077731140004665 06

```
36147761663001611607
76607761662001611710
77147761661001612011
77147761660001612112
77147761657777701413
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000

NUMBER OF NODES =    4991

SIZE OF EACH NODEIS        ONE COMPUTER WORD


NO. OF KEYWORDS IN THE QUERY

            2
QUERY
 8 463
RECORDS SATISFYING THE QUERY
RECORD              2
RECORD              3
RECORD              4
RECORD              6
RECORD              7
RECORD             17
RECORD             18
RECORD             22
RECORD             23
RECORD             31
RECORD             36
RECORD             40
RECORD             42
RECORD             44
RECORD             49
RECORD             51
RECORD             58
RECORD             60
RECORD             61
RECORD             70
RECORD             74
RECORD             84
RECORD             86
RECORD             95
RECORD             98
RECORD            103
RECORD            105
RECORD            111
RECORD            115
RECORD            116
```

RECORD 118
RECORD 120
RECORD 124
RECORD 130
RECORD 131
RECORD 132
RECORD 133
RECORD 136
RECORD 145
RECORD 146
RECORD 154
RECORD 155
RECORD 167
RECORD 171
RECORD 174
RECORD 195
RECORD 197
RECORD 199
RECORD 201
RECORD 203
RECORD 204
RECORD 206
RECORD 214
RECORD 215
RECORD 216
RECORD 219
RECORD 222
RECORD 224
RECORD 233
RECORD 241
RECORD 243
RECORD 244
RECORD 249
RECORD 253
RECORD 255
RECORD 256
RECORD 271
RECORD 272
RECORD 278
RECORD 279
RECORD 280
RECORD 284
RECORD 287
RECORD 288
RECORD 290
RECORD 294
RECORD 297
RECORD 300
RECORD 301
RECORD 304
RECORD 305
RECORD 307
RECORD 308
RECORD 311
RECORD 315
RECORD 320

```
RECORD          336
RECORD          341
RECORD          345
RECORD          346
RECORD          351
RECORD          360
RECORD          366
RECORD          374
RECORD          380
RECORD          388
RECORD          392
RECORD          398
RECORD          404
RECORD          410
RECORD          413
RECORD          414
RECORD          424
RECORD          437
RECORD          440
RECORD          441
RECORD          442
RECORD          447
RECORD          450
RECORD          452
RECORD          454
RECORD          458
RECORD          459
RECORD          463
RECORD          465
RECORD          467
RECORD          470
RECORD          482
RECORD          483
RECORD          484
RECORD          485
RECORD          487
RECORD          490
RECORD          492
RECORD          498
CPU TIME FOR ANSWERING THIS QUERY, IN SEC.

.4300000
END OF FILE SEARCH
```

```
3614776166300161160?
7660776166200161171?
7714776166100161201?
7714776166000161211?
7714776165777770141?
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
```

NUMBER OF NODES = 4991

SIZE OF EACH NODEIS ONE COMPUTER WORD

NO. OF KEYWORDS IN THE QUERY

3

QUERY

4 20 -70

RECORDS SATISFYING THE QUERY

RECORD 4
RECORD 34
RECORD 42
RECORD 55
RECORD 61
RECORD 85
RECORD 86
RECORD 92
RECORD 104
RECORD 128
RECORD 139
RECORD 142
RECORD 149
RECORD 154
RECORD 173
RECORD 176
RECORD 177
RECORD 186
RECORD 195
RECORD 202
RECORD 206
RECORD 207
RECORD 216
RECORD 224
RECORD 243
RECORD 266
RECORD 268
RECORD 272
RECORD 273
RECORD 292

```
RECORD          300
RECORD          308
RECORD          348
RECORD          350
RECORD          352
RECORD          355
RECORD          357
RECORD          363
RECORD          390
RECORD          393
RECORD          396
RECORD          401
RECORD          402
RECORD          405
RECORD          407
RECORD          413
RECORD          415
RECORD          427
RECORD          429
RECORD          433
RECORD          435
RECORD          443
RECORD          446
RECORD          449
RECORD          464
RECORD          465
RECORD          472
RECORD          482
RECORD          483
RECORD          487
RECORD          494
RECORD          495
CPU TIME FOR ANSWERING THIS QUERY, IN SEC.

.2680000
END OF FILE SEARCH


NO. OF KEYWORDS IN THE QUERY

          15
QUERY
-4   5   6 -25 -26  27  40 -41 -42 -49  50 -51 -58  59  60
RECORDS SATISFYING THE QUERY
RECORD          444
CPU TIME FOR ANSWERING THIS QUERY, IN SEC.

.0700000
END OF FILE SEARCH
```

```
3.61477616630016116O7
7660776166200161171O
771477616610016120 11
7714776166000161211 2
7714776165777770141 3
0000000000000000000
0000000000000000000
0000000000000000000
0000000000000000000
0000000000000000000
0000000000000000000
0000000000000000000
0000000000000000000
0000000000000000000
```

NUMBER OF NODES = 4991

SIZE OF EACH NODE IS ONE COMPUTER WORD

NO. OF KEYWORDS IN THE QUERY

10

QUERY

1 2 -3 -13 14 15 -28 29 30 58

RECORDS SATISFYING THE QUERY

RECORD 26

CPU TIME FOR ANSWERING THIS QUERY, IN SEC.

.0570000

END OF FILE SEARCH

```
3614776166300161160 7
7660776166200161171 0
7714776166100161201 1
7714776166000161211 2
7714776165777770141 3
0000000000000000000
0000000000000000000
0000000000000000000
0000000000000000000
0000000000000000000
0000000000000000000
0000000000000000000
0000000000000000000
0000000000000000000

NUMBER OF NODES =   4991

SIZE OF EACH NODEIS        ONE COMPUTER WORD


NO. OF KEYWORDS IN THE QUERY

           6
QUERY
-1   2   3 -82 -83 -84
RECORDS SATISFYING THE QUERY
RECORD            7
RECORD           41
RECORD           51
RECORD           95
RECORD          214
RECORD          297
RECORD          424
CPU TIME FOR ANSWERING THIS QJERY, IN SEC.

.1130000
END OF FILE SEARCH
```