NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us a poor photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de mauvaise qualité.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

COMPACT FILE ORGANIZATION

FOR

INFORMATION RETRIEVAL

Raymond Kernizan

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

May 1977

ABSTRACT

COMPACT FILE ORGANIZATION FOR INFORMATION RETRIEVAL

Raymond Kernizan

The objective of this thesis is to design a compact file organization for a document retrieval system. In such a system three main problems have to be solved 1) efficient use of storage space 2) reduction of execution time 3) transportability of the system.

In addition to these requirements, two features were considered to be extremely desirable: data independence and transparency of the programme. Data independence is desirable in order that the compression logic may be applied without change, to other files.

The file consists of both a serial file of complete bibliographic entries and an inverted file organized with respects to search keys, such as title words, subject terms, etc. Rapid search of the indexes to the catalog file is made by a method of double-hashing that allows front and/or rear truncation, which means that fragments can be used as query elements.

The file organization is similar to a paged-segmented memory and provides hierarchical storage where only the first level is in central memory. The access to each level is made by means of a hashing function. The compression scheme is based uppon Zipf's law which also predicts the proportion of data base text that appears in each level. A compression ratio of up to 38 percent is expected, and a ratio of 44 percent was obtained for texts chosen from the Communications of the Association for Computing Machinery.

ACKNOWLEDGEMENTS

My deepest thanks go to my supervisor, Prof. H.S. Heaps, without whose advice this thesis would not have existed. His guidance, suggestions, careful reading of drafts and helpful comments have been an invaluable aid throughout both the research and the writing process.

I am very grateful to Dr. T. Radhakrishnan for various discussions and advices, to Dr. W. Atwood for providing useful information about the PASCAL compiler, and to Carlo Locicero for his typeset program.

A special thanks to my parents for their moral and financial support during my studies, and to my wife Els for her encouragement and patience.

This project was supported by a research grant from the National Research Council of Canada and financial aid from the Computer Science Department of Concordia University.

POOR COPY

TABLE OF CONTENTS

POOR COPY

POOR COPY

TABLE OF TABLES

TABLE OF FIGURES

CHAPTER I

OVERVIEW AND OBJECTIVES

1.1 Historical Outline of Library Automation

The origins of library computerization may be traced to some of the engineering libraries that were newly established in the 1950's. Harley E. Tillit presented the first report on library computerization at the United States Naval Ordnance Test Station (NOTS), now the Naval Weapons Center. The report entitled "An experiment in information searching with the 701 calculator" (1) was presented at an IBM computation seminar in May 1954. The system was extended and improved in 1956, and a published report appeared in 1957 (2). Tillit subsequently published an evaluation of the system (3).

Tillit and his colleagues deserve much credit for their pioneer computerization of a subject information retrieval system. The application required considerable ingenuity because the IBM 701 did not have built-in character representation. Therefore it was necessary to develop subroutines that simulated character representation (4). Moreover, the 701 had unreliable electrostatic core memory. On some machines the mean time between failures was less than twenty minutes.

In September 1958 General Electric's Aircraft Gas Turbine Division used an IBM 704 computer and initiated a system (5) that was similar to the NOTS application. The General Electric System was an improvement over the then-existing NOTS system because it printed out author and title information for each selected report, as well as an abstract of the report. Like the NOTS system, however, the General Electric application provided only for boolean "AND" search logic.

The MEDLARS system (6) encompassed the first major departure in machine citation searching. The original MEDLARS had

two principal products: 1) Composition of a medical index (MEDICUS): and 2) machine searching of a huge file of journal article citations for production of recurrent or on-demand bibliographies. The system became operational in 1964. The MEDLARS system also provides for boolean "AND", "OR" and "NOT" search logic.

The next major development was CIALOG (7), an on-line system for machine subject searching of the NASA report file. Queries were entered from a remote terminal. The SUNY Biomedical Communication Network constitutes an important development in operation of machine subject searching and production of subject bibliographics of traditional library materials. The SUNY network began operation in the autumn of 1968 with nine participating libraries (8). Its principal innovation was the provision of on-line searches from remote terminals of the MEDLAR journal article file extended by the addition of book references. The SUNY network eliminates the two major disadvantages of the NOTS system, and all subsequent bath systems, in that it provides the user with an immediate reply to his search query.

In 1960 Bunnow prepared a report (9) in which he recommended a computerized retrieval system similar to the NOTS and General Electric systems but with the ability to produce catalog cards. The next development relating to catalog card production occured at the Air Force Laboratory Library, which began to produce cards mechanically in upper and lower case in 1963. A special computer-like device manipulated a single machine-readable cataloging record on paper tape to produce a complete set of card images punched on paper tape. This paper-tape product drove a mechanical typewriter that produced the card in upper and lower case. Two years later Yale University (10) began to produce catalog cards in upper and lower case directly on a high-speed computer printer.

The New England Library Information Network, NELINET,

demonstrated in a pilot operation in 1968 a batch processing technique servicing request from New England State University Libraries, via teletype terminals, for production of catalog card sets, books labels, and book pockets from a Marc-1 catalog data file. Also in 1968 the University of Chicago Library began operation of catalog card production with data being input remotely through terminals in the library, and with cards being printed in batches on a high speed computer printer.

Other fundamental library operations such as circulation, serials, acquisitions and standardization were also computerized (11).

This brief historical outline reveals two major trends in library automation. Firstly, there are those applications designed primarily to benefit the user, and secondly there is the employment of computers to perform repetitive routine library tasks such as catalog production, order and accounting procedures, serial control and circulation control.

1.2 Library File Organization

One of the problems that data processing professionals must consider is the organization of library files. These are some of the largest and most voluminous files that have to be organized, maintained and searched. They range in size from the National Catalog of the Library of Congress, which has over sixteen million records with an average of three hundred characters each, down to the hundreds of small college catalogs of 100,000 records. All such files are growing at a high rate. Since the turn of the century the university libraries have been growing exponentially and at present are doubling, on the average, every fifteen years (12).

Library files have certain common characteristics. First, as already noted, they are large. In the next ten or fifteen

years there will probably be several hundred libraries with holdings that each exceed one million volumes. Second, the records themselves are alphabetic and tend to be voluminous. They range in size from two hundred characters in an index journal, to three hundred characters for the standard catalog card, and up to two thousand characters for the abstract journal. Also, the file records are of variable length since the librarian cannot control the size of his inputs.

Records in a large catalog file are generally stable and not dynamic. If there is a new edition of a document then a new bibliographic record is made. If the old document is retained along with the new edition, the old catalog record is also retained. The record is discarded only if the document is discarded, and in the large research library this occurs very infrequently.

Each record item must have a number of different access points, since a single class or access point which everyone will accept is a practical impossibility. Furthermore a user wants to access files in open language; he will not use codes and will tolerate a minimum amount of training on methods to interrogate the file. He prefers to engage in dialogue with the file and also wants real-time response.

1.3 Randomization of Large Files

One of the well-known methods of randomizing a large file is to use a key-to-address conversion algorithm. This technique has been used by the Washington State University Library Acquisition Sub-system (13). The size of the file currently varies from approximately 12,000 to 15,000 items and has a capacity of 18,000 items. Over 40,000 items are added and purged annually. Each record consists of both fixed length fields and variable length fields. Records are blocked at 1,000 characters for file structuring purposes; however the

variable length information is treated as strings of characters with delimiters. The key to the file is a 16 character structure: six digits of the original purchase order (PO). Two digits of partial order and credit information, and eight digits containing the computed relative record address. Proper development of this key turns out to be the most important factor in achieving efficiency in both file access time and record density within the file.

The file resides on an IBM 2316 disk pack, which is part of an IBM 360 system running under OS. The I/O is performed by BDAM (basic direct access method) which uses the linear search. Thus the system locates the generated disk address, and if another record is found there it sequentially searches from that point forward until a vacant space is found and then stores the new record in that space. The sequential search is done by a hardware program in the I/O channel and proceeds at the rotational speed of the device on which the file resides. Similarly, when searching for a record the system locates the disk address and matches keys, if they do not match it sequentially searches forward from the address. Long sequential searches sharply degrade the operating efficiency of an on-line system.

In initial experimentation with the Washington State University file it was discovered that some records were 2500 disk positions away from their computed location. This seriously reduced response time to any terminals that were operating against those records. The necessity to develop a method for placing each record close to its calculated location became quite obvious.

The upper bound delay for a direct access read/write operation can be defined as the largest number of continiously occupied record locations in the file. The problem of minimizing the upper bound for a particular file is equivalent to

finding an algorithm which maps the keys in such a way that unoccupied locations are interspersed throughout the file space. One method of doing this is to increase the amount of space required for the files. This has been a traditional approach but is unsatisfactory because of its low efficiency in space utilization. The home address was computed as HA=(PO modulo prime number). When the file reached about 70 percent saturation, that is when 70 percent of the space allocated for the file was occupied by records, the method became unusable. Records were then located so far from their original addresses that terminal response time became degraded and batch process routines began to have significant increases in run time.

With no additional space available to expand the size of the file it became necessary to increase the record density within the existing file bound. The hash address was now chosen equal to [(PO mod prime number)+ ((300*PO) mod prime number)]/2. Again this scheme brought some relief, but the file continued to grow as the system was implemented and it became obvious that the procedure would also fail because of the occurrence of over crowded areas in the file.

1.4 Objectives

The objective of this thesis is to design a compact file organization for a document retrieval system. This is understood to be an information system which enables a user to:

1) search for any document related to a certain field. A document is represented as a set of elemental entities that contain information. Thus, consistent with this definition, a document might be a report, an abstract, or perhaps even a file card that contains key words and index phrases.
2) obtain further information about a document of which only the title is known.

POOR COPY

The information system is oriented toward two main categories of user:

1) those that are looking for a specific document, (title or author searching)
2) those that want to have more information about a specific document. (abstract searching)

As far as the user is concerned such a system has to be:

1) reliable
2) easy to use
3) fast.

This last factor is only significant when the system is designed for an on-line system. For such a system an answer received within 5 minutes following the query is considered to be acceptable and sufficiently fast in relation to a "human-scale" even though it could be considered to be slow in reference to a "computer-scale".

From the design point of view three main problems have to be solved:

A) efficient use of storage space
B) reduction of execution time
C) transportability of the system.

In addition to mandatory requirements involving compression ratio, core space and execution time, two features were considered to be extremely desirable: data independence and transparency of the programme.

Data independence is desirable in order that the compression logic may be applied without change to other files and to avoid change to the compression logic as the record undergoes frequent

minor modifications, such as by addition of new fields or by redefinition of field lengths and contents. Techniques such as differencing, decimal to binary, which amount to recoding values for individual data elements, are not data independent unless they can be table driven. If the routine can be applied to another file, or if it can be made to accomodate changes to record format and content by merely altering values in a table, the objective of data independence is achieved. Transparency to the program implies that no application program statements are required to invoque compression or decompression.

Such an attempt has already been made by Dimsdale and Heaps (14). They describe a file organization and design of an on-line catalog suitable for automation of a library of one million books. Rapid search of the indexes to the catalog file was made possibly by a method of virtual hash addressing (38), and storage of textual material in a compressed form allowed considerable reduction in storage costs.

The on-line catalog file consists of both a serial file of complete bibliographic entries and an inverted file organized with respect to search keys, such as title words, subject terms, author names, and call numbers. The advantages of such a two level structure have been pointed out by Warheit (15). A key may be operated on by a hashing function which transforms it into a pointer to an entry in a hash table file. This file contains pointers to both a dictionary file of title words and an inverted index stored in a compressed form. Entries within the compressed inverted index serve as pointers to the catalog file of complete bibliographic entries.

CHAPTER 2

SPACE AND TIME CONSIDERATIONS

2.1 Priorities and Trade-Offs

Taking into account the difference in scale between human time and computer time as pointed out in the previous chapter it is considered that reduction of execution time is not the prime factor in design of a retrieval system. However data compression is essential. This is believed to be so because a document retrieval system is one of the rare information systems in which the information is never, or rarely, deleted. In contrast new entries are frequently added, particularly if the majority of document items represent either articles in periodicals or books and reports whose numbers increase steadily because of new acquisitions.

It is thus believed most important to give special consideration to file compaction and data compression problems. It is essential to avoid a character by character coding that could be very expensive and in some cases physically impossible.

The proposed compression scheme is based on statistical encoding with particular emphasis on the implications of Zipf's law. Thus the code assigned to a word is a function of its frequency of occurrence in the entire data base, the most frequent words receiving the codes of minimum length. The code, which also indicated both the position in a hierarchy of dictionaries at which a descriptor is stored and the level of the particular dictionary in the hierarchy, may be compared to an address in a paged memory. The address computation is done by mean of a hashing function applied to fragments of a descriptor.

Another important consideration is that the system be easy to use. This implies the use of a very simple, but

powerfull, query language. Each query is a general statement that describes the information that is sought. It is a formalized expression of a request for information. In general it is a string of words connected by logic operations.

2.2 Historical

One of the best well-known automatic document retrieval systems is the SMART system (16). The system is characterized by the fact that it allows use of several hundred different methods for analysis of documents and specification of search requests. This feature is used in the retrieval process by leaving the exact sequence of operations initially unspecified, and adapting the search strategy to the needs of individual users. The system may therefore be used, not only to stimulate an actual operating environment, but also to test the effectiveness of several alternate processing methods.

The particuliarity of the SMART system resides in the fact that in retrieval of a document the procedures may extend the original request based on simple word matching procedure by generating more effective content indicators to identify documents and search requests. This is accomplished in part by generating word stems from the original word forms, by introducing synonym dictionaries to lessen the effects of vocabulary variations, and most importantly by identifying relations between certain words to be used as content indicators in conjunction with the surrounding words.

Stored document and search requests are processed without any prior manual analysis by one of several hundred possible methods. Those documents which most nearly match a given search request can be processed first in a standard mode after which the user is free to analyse his output and, depending on his further requirements, he may specify to have a reprocessing of the request under new conditions.

SMART is thus designed to serve as a reasonable prototype for fully automatic document retrieval. The following remarks summarize the principal advantages that are claimed for the system.

(1) the information analysis is believed to be sufficiently deep and refined to ensure the identification of most relevant material in answer to most search requests.

(2) the varying needs of the individual are recognized by enabling each user to call on many different text processing modes, and by choosing a suitable sequence of procedures that it is hoped will eventually lead to satisfactory retrieval performance.

(3) the system can serve as a means for evaluating the effectiveness of a large variety of automatic procedures of question analysis in that the same search request can be processed against the same document collection in many different ways and the results compared. However, implementation of the SMART system has been carried out for very small experimental data bases and not for a large operational system.

A document retrieval system must have some means of recording the subject matter of each document in its data base. Some systems store the actual text words, while others store keywords or similar content indicators. The SMART system uses concept number for this purpose, each number indicating that a certain word appears in the document. Two advantages are apparent. Firstly, a concept number can be held in a fixed size storage element to allow faster processing than if variable size keywords are used. Secondly, the amount of storage required to hold a concept number is less than that needed for most text words and so storage space is used more efficiently.

SMART must be able to find the concept numbers for the words in any document or query. This is done by a dictionary lookup. There are two reasons why the lookup must be rapid. For text lookup a slow scheme is costly because of the large number of words to be processed. Also for handling user queries in an on-line system a slow lookup adds to the user response time.

2.2.1 Storage By Use Of Dictionary

Storage space is also an important consideration. Even for moderate size subject areas the dictionary can become quite large, either too large for computer main memory or so large that the operation of the rest of the retrieval system is penalyzed. In most cases a certain amount of core storage is alloted to the dictionary, and the lookup scheme must do the best possible job within this allotment. This usually means that the overhead for the scheme must be kept as low as possible in order that a large portion of the alloted core is available to hold dictionary words. The rest of the dictionary is placed in auxiliary storage and parts of it are brought in as needed. Obviously the number of accesses to auxiliary storage must be minimized.

The proposed document retrieval system is not as smart as SMART because there is no thesaurus and only a keyword matching procedure is used. The effectiveness of the search will be determined by the accuracy of the query structure. In this regard it is more similar to CAN/OLE which is a reference retrieval system. CAN/OLE (Canadian On-Line Enquiry) is an on-line information retrieval system. The system was designed and is operated by the Canadian Institute for Scientific and Technical Information (CISTI) and the National Research Council Computer Center, for the retrospective searching of large bibliographical reference files.

2.2.2 Storage Based On Polarized Distance

Document retrieval models have been investigated by various authors, among them Chien and Mark (17) presented a document storage method based on polarized distance in which a document retrieval system is defined to consist of an ordered index term set (with T terms), a document set stored on mass memory, and a threshold retrieval or matching function F.

A threshold can be interpreted as follows: "A logic operator such that, if P,Q,R,S,... are statements, then the threshold will be true if at least N statements are true; false otherwise."

This logic operator is a generalization of "AND" and "OR". "AND" is a threshold with a value of N equal to the number of arguments whereas "OR" is a threshold with a value of N equal to one. A threshold is sometimes called a majority.

Each document in the system has associated with it a binary index (descriptor) vector denoted by Y which indicates which subset of terms is used to index or tag the document. A query is represented by a binary term vector X which the user composes to represent his area of interest. In a sense X represents the index vector of the user's "ideal document". Upon receipt of the query X the system evaluates F(X,Y), which is a predetermined function, for each document descriptor vector Y in the collection and retrieves or sends to the user all documents which have index vectors that satisfy $F(X,Y) \leq$ SMALLT (a threshold). The set of terms for which $F(X,Y) \leq$ SMALLT is the retrieval set, and the non-retrieval set is the one for which $F(X,Y) \leq$ SMALLT.

According to this definition two main tasks have to be considered:

1) space allocation
2) query processing.

The storage allocation procedure is very simple and efficient. In a sequential file organization procedure (e.g. tape) the method is to store documents in a linear array according to the vectorial norm of Y, where Y is a document index vector. Thus, the documents with Norm(Y) = 1 (if any) are stored in bucket 1 in the array followed by documents with Norm(Y) = 2 in bucket 2, etc. The starting location of each bucket in the memory is recorded. The mathematical background of the storage strategy and the retrieval policy is discussed by Chien and Mark (17).

It is believed that search methods based on the mathematical properties of term matching simplify the key-to-address computation in comparison to the clustering or coding techniques.

2.3 Polarized Distance And Clustering

The polarized distance is closely related to the key-to-address transformation. This relationship is expressed in the following sections.

Let the set of all possible keys be denoted by set (K). The keys are strings of length N of symbols, each symbol being taken from an alphabet of 2**Q symbols, where Q is a number of bits per characters. The keys that appear in any particular file form a small subset(S) of the set(K). It is further assumed that the memory has M addresses numbered from 1 to M that form a set denoted by set(A). Obviously the number of elements of any set(S) must not exceed M which is the number of elements in set(A).

An examination of actual key sets S reveals that a typical characteristic is the occurence of clusters. For example, ABCD and ACBE are distance 3 from each other in any alphabet which contains A,B,C,D and E. This distance is equal to 3 because three of the four positions of the strings do not match.

This distance definition, however, depends on the representation. For example, if A,B,C,D,E are respectively coded as 000, 001, 010, 011, 100 so that ABCD becomes 000001010011 and ACBE becomes 000010001100 then the distance of the two keys in his two-symbol representation is 7 and not 3. For this reason a fixed representation is used, that is, a fixed Q. A cluster can be defined now as a set of keys which are near to each other. To be more precise we define a cluster of diameter D as any set of keys, in which the maximum distance between pairs is D. For example, the following set of names is a cluster of diameter 4 in any alphabet containing A,B,C,N,O, R,U,W

BR AU N A B

BR AU N A C

BR AU N B C

BR OW N B C

BR OW,N B A

Polarized distance has been used to solve the clustering problem. The method involves finding a transformation which destroys the clusters in the set(S) and disperses the elements of a cluster among the storage buckets. The transformation will map the elements of any cluster whose diameter is less than D into a different address and D should be as large as possible. If the maximal D is denoted by MD, the only restriction in any input set S is that no cluster exceeds diameter MD.

The aim of the coding method is to partition the given set K into M subsets in such a way that the elements in each subset are at least distance MD away from each other. By numbering these subsets from 1 to M, and considering the

transformation which maps every key into the number of the subset containing it, there results a transformation which maps any two keys, that are at a distance less than MD from each other, into different numbers.

Set(K) and set(A) are considered as vector spaces over the field 2**Q elements and have dimensions N and M respectively. The symbols of the alphabet are considered as elements of a field, one of them is the 0 of the field..

A linear transformation from a key (A1,...,AN) to an address (P1,...,PM) can be of the form

$$\sum_{J=1}^{N} Tij*Aj=Pi \qquad \text{for } I=1,.....,M$$

here Tij is the required transformation matrix and it has to have M rows and N columns.

The distance between every pair of keys mapped into the same address by the transformation matrix Tij is larger than MD if, and only if, every MD columns of the matrix are independent. This implies in particular that MD has to be less than or equal to M. This theorem has been proved by Schay and Raver (18).

2.4 Conclusion

For the clustering retrieval method all documents which are close to each other in the index space are put in the same bucket. Then a representative or centroid vector is generated to represent the bucket's content. Retrieval is then a two-part matching process. First, the query vector is matched against the representative vectors, and then for these buckets which have a high correlation to the query an exhaustive search

is made. The clustering method has the drawback that a document can match the query but be in a low correlation bucket and consequently can be missed during the retrieval process.

Since the polarized distance method requires documentary information to be stored only once it allows use of less memory than with coding and inverted filing schemes. Documents are filed independently of both the matching functions and the threshold can be used to interrogate the file. Finally, the size of the file can be easily increased so that larger files can be partitioned into smaller buckets for faster searches at only a slight expense in indexing overhead. However, it is dictionary storage that is assumed in the present thesis.

CHAPTER 3

SOME HASH CODING SCHEMES

3.1 Definition And Properties Of Hashing

In the proposed retrieval system the entire file organization is based on the use of a hashing scheme. A hashing method involves a hash code, or a key-to-address transformation, which will be denoted by H. If K is an arbitrary key then H(K) is an address. Specifically, H(K) is the address of some position in a table, known as a hash table, at which it is intended to store the record whose key is K. If at some later time it is desired to search for the record whose key is K then all that is necessary is to calculate H(K) again. In most instances the desired record may be located immediately without any need for a sequence of comparisons; this is the property that makes hashing methods so useful when dealing with large data bases.

The only time that a record cannot be stored at its home address H(K), where K is the key in that record, is when two or more records have the same home address. In practice there is no way that this can be prevented from happening because there are normally several orders of magnitude more possible keys than there are possible home addresses.

The phenomenon of two records having the same home address is called collision, and the records involved are often called synonyms. The possibility of collision, even if slight, is the chief problem that arises in use of hash table methods.

A similar phenomenon to collisions is clustering in which certain records are not synonyms but have hash codes in sequence. The result is that if the collision is handled by use of a linear search method of treating collisions the search progresses

through addresses ingremented by a multiple of an increment M, E.G. K+M, K+2*M, etc. K' = K+2*M for two hash addresses, K' and K. Thus the linear search method extends the size of the set of entries for which the table must be searched. Sometimes the cluster is generated by the hash function itself; this is called primary clustering. But sometimes it is generated by the collision handling algorithm; in this case it is called secondary clustering.

It is necessary to deal with three important problems as follows:

a) choice of a hash code that possesses the following attributes:

*) the algorithm should be simple. This leads to a short, quickly written, easily understood procedure.

**) the algorithm should generate an efficient, exhaustive sequence of addresses. Every location should be probed exactly once before the table is declared full.

***) the time required per probe should be minimal.

****) the average number of probes per lookup should be minimal.

b) the data should be compressed through representation by an efficient code that will minimize the storage requirements and allow relatively simple and fast access.

c) the file organization should minimize the number of disc-accesses, hence eliminating the problem of "trashing".

Among the attributes of a good hashing function, two are of particular importance and may be considered in terms of the

following two basic parameters:

1) the average time required to insert a new entry in the table

2) the average time for retrieval of an existing element of the table.

The values of the above two parameters are dependent on:

N: table size

M: total number of entries already used.

P: total number of free entries. (N-M)

T1: time required to compute the hashing function.

T2: time required to verify the presence of an entry in the table.

Further details of the dependence are discussed in the following two sections.

3.2 Average Insertion Time

Let F(N,M) be the average time required to introduce a new entry in the table. If this table is completely empty (M=0) the new entry associated with the hash value will be certainly empty and so

$$F(N,0) = T1+T2$$

Now suppose that there is one occupied entry (M=1). The home address generated by the hashing function has a probability of (N-1)/N of being free and it is then necessary to check in

only a single entry in the table; the total time required for both operations is T1+T2. But the probability of having a collision is 1/N, in which instance it is necessary to do two succesive probes in order to find a free space, the time required for insertion is T1 + 2*T2. Thus

$$F(N,1)=[((N-1)/N)*(T1+T2)] + [(1/N)*(T1+2*T2)]$$

$$=T1 + [(N+1)/N)]*T2$$

If there are two elements in the table (M=2) the probability of hashing into a free space is (N-2)/N in which case the time is still (T1+T2). The probability of a collision is 2/N and the situation may be considered similar to that in which there is one occupied entry in a table of size N-1. This assumption is true if it is presumed that the hashing function generates random numbers uniformly. Since it is not necessary to compute a new hash value the average time to access the table of size N-1 is (N/N-1)*T2, and thus

$$F(N,2)=[((N-2)/N)*(T1+T2)] + [(2/N)*(T1+T2+(N/N-1)*T2)]$$

$$=T1+((N+1)/(N-1))*T2$$

$$= T1+((N+1)/((N-2)+1))*T2$$

The above formulas for F(N,0), F(N,1), and F(N,2) are special cases of the general formula

$$F(N,K) = T1 + [(N+1)/(N-K+1)] * T2$$

which may be proved by induction as follows by supposing the formula to be true for K=0,1,...,(M-1)

Suppose M<N entries are occupied. There is a probability of (N-M)/N that the entry determined by the hash value is empty.

In this case the average time for insertion is T1+T2. But if the entry is already used, and this occurs with probability of M/N, then after time T=T1+T2 exactly M-1 elements have been stored in a table of size (N-1), and storage of the remaining element will require an additional time of F(N-1,M-1)-T1 = N/((N-M)+1). It thus follows that

$$F(N,M) = [((N-M)/N)*(T1+T2)] + [(M/N)*(T1+T2+((N/((N-M)+1))*T2)]$$

$$=T1+[(N+1)/((N-M+1)]*T2$$

The average time for insertion of a new element may thus be written in the form

$$T1+[(1+(1/N))/(K+(1/N))]*T2$$

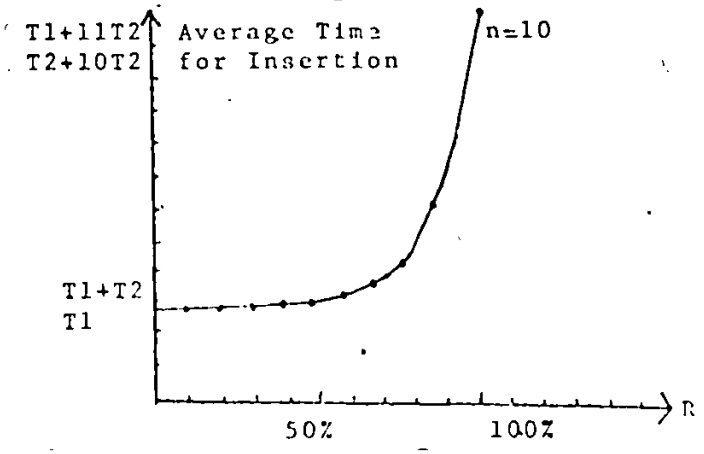
where K=1-M/N denotes the proportion of free space. In terms of the loading factor R=1-K the expression becomes

$$T1+[(1+(1/N))/(1-R+1/N))]*T2$$

which for large values of N may be approximated as

$$T1+(T2/(1-R)).$$

The variation of average insertion time with loading factor R is shown in Fig 1.



3.3 Average Retrieval Time

Consider a table of N positions that already contains M.GE.1 elements. The time required to retrieve one of these elements is equal to the time required to store it. If the particular entry is the first one that has been stored the retrieval time is equal to T1+T2, if it is the second the time is T1+((N+1)/N)*T2 and so forth. The average of all the possible cases is thus:

$$G(N,M)=(1/M*\sum_{K=1}^{M}F(N,K)$$

$$=T1+T2*\ ((N+1)/M)*\sum_{K=1}^{M}(1/(N-M+1))$$

The value of the summation may be estimated by noting that

$$1/(K+1) \lt \int_{K}^{K+1} DX/X \lt 1/K$$

and hence

$$\int_{K}^{K+1} DX/X \lt 1/K \lt \int_{K-1}^{K} DX/X$$

thus

$$LN((N+1)/(N-M+1))= \int_{N-M+1}^{N+1} DX/X \lt \sum_{K=1}^{M} 1/(N-K+1) \lt \int_{N-M}^{N} DX/X = LN(N/(N-M))$$

and so the average retrieval time G(N,M) satisfies the inequality

$$T1+T.2*[((N+1)/M)*LN((N+1)/(N-M+1))] \lt G(N,M) \lt$$

$$T1+T2*[((N+1)/M)*LN(N/(N-M))]$$

for large N the expression may be written in the form

$$T1-T2*[LN(1-R)]/R$$

where R is the loading factor. As the loading factor increases the average retrieval time increases as summarized in table 12.

R	1/1-R	$\frac{Ln(1-R)}{R}$
0.00	1.000	1.000
0.25	1.333	1.151
0.50	2.000	1.386
0.65	2.857	1.615
0.80	5.000	2.012
0.90	10.000	2.558
0.95	20.000	3:3368
0.99	100.000	4.652

Table 12

The results described in the present and previous sections are well-known, and there exists a considerable amount of literature concerned with various aspects of hash storage.

3.4 Survey Of Some Proposed Hashing Schemes

In the early 60's Buccholz (19) presented the concept of hash coding as a method for key-to-address transformation. Different methods such as division, folding, and truncation were presented, as were the most frequently used overflow handling methods. The notion of table hierarchy was introduced.

In 1968 Maurer (20) popularized a method that has been suggested by Mealy (21). This method called "division hash code" was very different from the previous hash methods called

logical and multiplicative schemes that involved the calculation of a k-bit field assumed to be a random integer between 0 and 2**k-1.

Let K be the integer whose value represents the number of bits required to code the number of positions in the hash table. The logical methods use the "exclusive OR" function applied to certain parts of the word to be hashed (for example the first 4 characters). Multiplicative methods consist of multiplying the number to be hashed either by itself or by a constant, and taking some k-bit field of the result. Logical methods are faster than multiplicative methods except on machines with very fast multiplication. Logical methods, however, sometimes fail spectacularly to give randomly distributed hash codes. For exemple with a logical method that chooses the 4 first characters then words like compute, computer, and computerized, will produce the same hash code.

Logical and multiplicative methods were very restrictive because the sizes of the tables were restricted to values of the form 2**k, while the division hash code allowed the hash table to have almost any length. Also a new method for handling collisions was introduced. When a collision occurs the new location is not determined by a linear method but by a quadratic function (K+A*I+B*I**2) modulo the table size. The parameters K, A,B are fixed, and I is a positive integer that is used for purposes of incrementing.

In the meantime Morris (22) made a review of some logical and multiplicative methods. He proposed a pseudo-random number generator for resolving collision problems by use of a random probing. Theoretical relations between the average number of probes necessary to retrieve an item and the fraction (alpha) of the table which is occupied (loading factor) were derived for the random probing, the linear probing, and direct chaining, methods. Two powerfull tools, the scatter index tables and

the virtual scatter tables, and some of their utilizations were also presented.

When Morris introduced the quadratic search method, it was with the aim of resolving the "primary" clustering problem. If H(K) denotes the calculated address for the key K then for the quadratic search the equality:

$$Hi\ (K) = Hj\ (K) \quad i \neq j$$

implies that

$$(H_{i+L}\ (K),\ L=1,2,3$$

and

$$H_{j+L}\ (K'),\ L=1,2,3$$

do not trace out the same sequence. This is contrary to the situation in the linear search where $H_{i+L}(K)=H_{j+L}(K')$ for every L.

However, another kind of clustering still remains. If in a quadratic search $Hi\ (K) = Hi\ (K')$ then

$$H_{i+L}\ (K) = H_{i+L}\ (K')\ L=1,2,3...$$

in particular, two keys which initially have the same location proceed to trace through the same sequence of locations; that is, those keys which hash initially to the same location will contribute to formation of a cluster.

To avoid such a situation Bell (23) suggested use of quadratic quotient hashing which is an extension of the quadratic search form

$$Hi\ (K) = Ho(K) + A*I + B*I**2$$

to become

$$Hi(K) = Ho(K) + A*I + B(K)*I**2.$$

the function B(K) is chosen as the integer value of the quotient (K/N) used to compute Ho.

An evaluation of the efficiency showed that the quadratic quotient is superior to quadratic search, and the incremental cost is very low and could be zero on some machines. But the quadratic search and the quadratic quotient method have certain disadvantages. When the table size is a power of 2 the period of a quadratic search is usually too small for effective use, and when the table size is a prime number a quadratic search always covers exactly half of the table because the method is based on quadratic residues. A quadratic residue for a prime number P is an integer R, where R≠P, for which there exists an integer I such that (I**2 - R) is divisible by P, i.e. P/(I**2-R). However it is well-known that for a given prime number P and for all integers I there exist only (P-1)/2 distinct quadratic residues, where R=0 is excluded from the set of residues.

To avoid the above disadvantages Radke (24) suggested some modifications to the standard algorithm. First he suggested that the size P of the table be chosen in such a way that P is a prime number of the form (4*K + 3) for some integer K. When a collision occurs the search is first made by the quadratic function:

a) A = (I**2 + K)(MOD P)
and if this function fails then the search is made by the quadratic

b) B = P + 2*K - (I**2 - K)(MOD P)
Radke insisted that if the table size had to be a power of 2 then P should be chosen so that P≠2**N.

POOR COPY

Hopgood and Davenport (25) refute this assumption. They reconsidered the quadratic hash function and tailored the collision handling algorithm for a table whose size is a power of 2. The quadratic hash method avoids clustering by defining the Ith position in a sequence as

K+A*I+B*(I**2)

the next position in a sequence then depends on the length of the sequence so that random collisions of two sequences do not cause them to combine. The computation required for the quadratic search method can be reduced by defining R= A+B and Q = 2*B so that the algorithm becomes

1) calculate K = I(K), J=R

2) if the Kth position is empty or contains K then the search is concluded.

3) otherwise set K = K+J, J=J+Q and repeat 2.

They disprove the affirmation of Maurer that when M is a power of 2 the period of a quadratic search is usually too small for effective use. However, in the special case that Q=1 the period of search is M-R+1 (25). Consequently with R=1 the complete table is searched. The period of search in this case is considerably better than the case where M is prime for which the period is M/2. The algorithm therefore takes the simple form

1) calculate K = I(K), J=R

2) if the Kth position is empty or contains K the search is concluded.

3) otherwise set K=K+J, J=J+1 and repeat 2.

An unusual property of the method is that the period of search is reached when the Ith and (I+1)th entries are the same. Thus the sequence repeats when j=m.

Hopgood and Davenport gave an expression for the average length of search in use of their method and made some comparisons between the linear search, the usual quadratic search, and the quadratic quotient search. Fixing R=1 did not alter the properties of the quadratic search.

When the length of the table is a prime number of the form 4*N+3 the whole table may be accessed for the original entry point. Collin Day (26) proposed the following new algorithm that searches the entire table.

a) set I to -P.

b) hash the datum to value J.

c) if location J is not filled, make the entry and stop. Otherwise set I = I+2:

d) set J = (J + ABS(I)) MOD P .

e) if I<P return to (c), otherwise the table is full.

In this method, a test for a location being filled is made only once. The increment ABS(I) is never larger than (P-2), and is always positive.

Although the quadratic search methods eliminate the secondary clustering they do not guarantee a full table lookup. Furthermore there is some restriction on the size of the table (4N+3) if a full table search is required. On the other hand the linear search produces so many collisions that the efficiency of the hash function becomes very poor. Bell and Kaman (27)

tried to eliminate those inconveniences by use of the linear quotient hash code. The method was presented as simple, efficient, and exhaustive, while needing little time per probe and using few probes per lookup. The home address is computed by a division, and the search algorithm is through use of linear probing, but the step for the jump is equal to Q(K) which is the integer value of the quotient. The table is full when N probes have been made. The method was compared to the quadratic quotient and it seems that the number of collisions is increased by 3 however, the time per probe for the new method is reduced by 10 and the result is a net saving of 7 percent.

Luccio (28) has presented on alternative a method of making a linear search by use of a function of a key k. He introduced a method called "weighted increment linear search" for scatter storage tables. The method applies to tables of size N=2**P; it allows a full table searching, and practically eliminates primary clustering at a very low-cost. The modification made by Luccio was to weight the increment linear search. The formula for computation of a home address becomes

$$Hi(K) = [Ho(K)+Q(K)* I] (MOD\ N)\quad I=1,2...$$

where Q(K) is an integer function of the key. Then the increment step is generally different for different keys.

To ensure that the entire table is searched, for any k, Q(K) must be coprime with n. This is achieved by setting

$$Q(K) = (2*P(K)+1) \quad (MOD\ N)$$

where P(K) is an integer constant that can be extracted from the key by any hashing method. In particular, the hash address Ho(K) can be substituted for P(K) and the increment law results in the form

$$Hi(K) = [Ho(K)+[2Ho(K)+1] \ (MOD\ N*I)] \ (MOD\ N) \quad I=1,2..$$

this formula can be evaluated just once, and any address H[I+1](K) in the sequence computed from the preceding one by one addition

$$H[i+1](K) = [Hi(K) + Q(K)] \ (MOD\ N)$$

All these hash storage methods were applied by Lum, Yuen, Dodd (29) to a set of existing files. As each method was applied to a particular file, the load factor (alpha) and the bucket size was varied over a wide range. In addition, appropriate variables pertinent only to a specific method were given different values. The performance of each method was summarized in terms of the number of accesses required to retrieve a record and the number of overflow records created by a transformation. Pecularities of each method were discussed and practical guidelines obtained from the results were stated.

3.5 Conclusion

In conclusion, one can say that no hashing method is ideal, because too many parameters, like the vocabulary of the data base, the size of the hash table, the length of the computer word, the time required by a specific computer to perform a basic operation and so on, are involved. Before choosing any particular method, a simulation should be made on a sample of the specific data base.

CHAPTER 4

DATA COMPRESSION

4.1 Definition

A few terms are needed to define the problem of data compression precisely. As suggested by Rubin (30), the body of text to be encoded is called the input string (IS). The semantic units of the input string are called characters, and may consist of Roman letters, light intensities, or pulse amplitudes. The codes which represent the characters will be called input codes, and the correspondence between the input characters and the input codes will be called the input representation. The input representation might be paper tape code, greyness level codes, or decimal digit strings.

Encoding consists of dividing the input string into substrings called input groups. Each input group is replaced by an output code. The correspondence between input groups and output codes is called the output representation (OR).

The goal of encoding is to obtain as compressed an output as possible. One could simply let the output code 0 represents the input group consisting of the entire input string. Obviously, there is no gain with this procedure, as it would still be necessary to show that this code corresponded to that particular input group. The length of the output is therefore regarded as including both the string of output codes, and the output representation. The measure of compression which is used is thus

$$(LEN(IS)-[LEN(OS)+LEN(OR)])/(LEN(IS))$$

expressed as a percentage.

4.2 Advantages Of Data Compression

Data compression techniques are designed to reduce

storage space for data files at the price of the increased CPU activity needed for compression and decompression. As CPU time becomes cheaper relative to the cost of external storage devices compression methods become increasingly attractive for dealing with large files. The most effective use of compression is in application to files in which the main consideration is minimization of physical storage space. Reduction of storage can be achieved by means of compaction which is used to describe any technique which reduces the size of the physical representation of the data while preserving a subset of the information deemed to be relevant information. Reduction may also be by compression of data, the term compression being used to describe a compaction technique which is completly reversible.

Data compression is of interest in data processing because it offers cost savings and the potential for increased capacity in mass storage devices, channels, and communications lines. By use of data compression we can avoid costs for rental and maintenance costs for disks or drums and perhaps additional control units or channel or site expansion required to house the new hardware.

Data compression is made by means of a compressor which is a software routine or hardware device which accepts a string of bits or characters of data and transforms it into a shorter string. A decompressor is a routine device which transforms the compressed string back to its original uncompressed form.

A variety of data design techniques resemble compression; however, they are not included in the above definition since they are applied at the time the data record is designed, and not inside a hardware or software "black box". These data design techniques included header-trailer techniques, packed fields, dual purpose fields, decimal to binary conversion. Abbreviating or devising coded values for individual data

elements, field differencing, etc. Data compression may take place independently of, and in addition to, field level data coding and other space-saving data design techniques. Other important compressor techniques like null suppression and pattern substitution have been surveyed by Ruth and Kreutzer in (31).

"Null Suppression" is a class of compressors which includes a wide variety of routines which supress zeros, blanks, or both. These are the simplest of all compression routines. They are widely used in business data processing, often to exclusion of any other techniques. They take advantage of of the most prominent characteristics of business data files, that is, the prevelence of blanks and zeros. They are simple in concept, easy to implement and usually achieve a reasonable degree of compression at a relatively low cost in CPU time. Null suppression can be packaged easily as a generalized routine that can apply to many different files and is routinely supplied in many vendor implementation packages, especially for data transmission to peripheral devices, such as printers. On the negative side, null suppression does not usually achieve as high a compression ratio as some other techniques.

One of the many ways of implementing null suppression is the Common Run Length Technique. A punctuation mark is inserted to indicate a run of zeros, and is followed by a number to indicate the length of the run. #N is substitued in the data string where a string of null characters longer than two occurs, N representing the number of zeros or blanks replaced, minus two. For example

Original Data: AE10000X000456000000N00000

Compressed Data: AE1#2X#1456#4N#3

Another compression technique is the pattern substitution.

Business data files often contain stereotyped patterns that occur in record after record. These may include numeric and alphabetic information combined with, or in addition to, the inevitable strings of zeros and blanks.

The basic approach to compression used here is to divide the text string into substrings, and to replace each substring by a code; the simplest such scheme is to choose strings of identical characters and replace them by a repetition count. This allows the encoding of only a small portion of the possible strings in any file.

Another simple approach is to choose substrings, typically character pairs, on the basis of standard English character frequencies. This is suitable for english prose, computer source languages or name and address data. The restriction to two-character substrings severely limits the degree of compression which can be obtained.

A third approach is to choose words or phrases as suggested by Kusmiss (32) and Hagamen (33). Again, this limits the choices of substrings for which codes can be substituted. It is suited only to prose files and sometimes to source program data.

The compressor can be built to scan the data and build its own pattern table or the table can be prepared in advance. The pattern table can be stored or transmitted with the compressed data, or it can be built in as a permanent part of both compressor or decompressor. As with null suppression, it is convenient to use unused characters from the character set as substitute values is these are available.

Pattern substitution achieves a greater compression ratio than null suppression, since patterns in addition to strings of zeros and blanks contribute to the compression. Efficiency

POOR COPY

is a serious problem in pattern substitution compressors since many comparisons must be made between the data and the pattern table in order to identify a pattern. Patterns are of various lengths, and may be subsets of other patterns. The pattern table may be organized in tree form to reduce the time required to identify a pattern. An example of pattern substitution technique is the following:

Original Data

AE10004MFQ00000F320006BCX4

AE20000BDF00000F300000BCX1

AE3D002RBA00000F301214BCX7

AE40006MDC00000F373000BCX4

Pattern Table

AE = #

000 = $

00000F3 = %

BCX = "

Compressed Data

#1$4MFQ%2$6"4

#2$0BDF%$00"1

#3$2RBA%01214"7

#4$6MDC%73$"4

Tree representation of the pattern table.

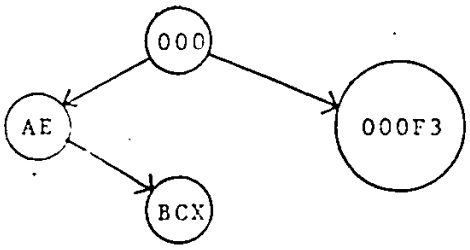


Fig. 9

4.3 Some Compression Schemes

Gottlieb, Hagerth, Lehot, and Rabinowitz, (34) discussed several techniques of data compression, such as

a) front compression/rear compaction.

b) compression by differencing.

c) statistical encoding.

A good example of compaction is the front-compression/rear-compaction scheme applied to a sequence of sorted keys. This compaction scheme results in a very compact first level directory in which are stored only those portions of a key (K) that are

- not identical to the previous key,

- necessary to make key unique.

details are described below.

4.3.1 Front Compression

The leading bits of a key which are identical to the

leading bits of the previous key constitute the "FRS" (front redundant string) and need not be repeated. The FRS is expanded to include one extra bit, since it follows automatically that if the N bits of a key are the initial repeated string, then the (N+1)st bit must be different. Instead of the FRS itself, a number can be stored to specify the length of the FRS. Storage of this number requires a field of bits of length equal only to the logarithm (base 2) of the length of the key. For example, if M is the length of the key, say M=32 bits, then the length of FRS cannot exceed M; hence, the number of bits needed to express the FRS-length is $\log_2(M) =$ 5 bits.

4.3.2 Rear Compaction

Unlike front compression, which suppresses some redundancy but does not really reduce the information provided one knows the previous key, the use of rear compaction will delete an "RRS" (rear redundant string). An RRS is composed of those right most bits of a key which are not necessary to uniquely distinguish this key from all previous keys in the particular sequence to be rear-compacted. One way of finding the RRS can be described as follows: (34) "Given a set of keys, some of which may be identical, in order to find the RRS of a key (K) it is enough to look at the previous key (P) and the following key (A) in the sorted sequence; i.e., the RRS for K relative to the whole set of keys is identical to the RRS for K relative to the set of 3 keys P, K, A."

The useful string (US) is what is left of the key after the FRS and the RRS have been removed. For example consider a sequence of three keys P,K,A as follows:

key (I-1): P=10101010101010101010

FRS = 1010101 US = 01010 RRS = 10101

key(I): K=10101011010101010

key(I+I): A=10101011010111011

key(I) will be coded as (8) 01010, where 8 is the size of the front redundant string (FRS) and "01010" is the useful string left over after front and rear compaction.

Note that the last bit, and only the last bit, of the FRS differs from the corresponding bit of the previous record. Note also that the FRS of key #I is sufficient to distinguish it from key #(I-1), but that the 5 bits of the useful string (US) are needed to distinguish key I from key #(I+1).

Another way of considering rear compaction is to define the useful string of key #I as follows: "if FRS of key #(I+I) contains the FRS of key #I then the useful string of key #I is the string obtained by deleting the FRS of key #I from the FRS of key #(I+1); otherwise it is null".

If the keys are regarded as binary numbers of N bits each, then the compacted key #I will occupy $[(\text{Log}_2([I+1]-K[I]))] + [\text{Log}_2(K[I]-K[I-1])] + 2*[\text{Log}_2(N)]$ bits in the first case and $2*[\text{Log}_2(N)]$ bits in the second case.

4.3.3 Compression by Differencing

The term differencing is used to describe techniques in which the current record is compared to a pattern record and is replaced by the difference between the two records. The technique is particularly successful for large records of alphabetic characters where most corresponding fields in different records are the same (or even blank or zero), such as dates, social insurance numbers, part number, etc. Also compression is often improved by sorting the file on the largest field. For example, consider a series of words such

POOR COPY

as computed, computer, computerized. These data items can be represented as comput, ed, er, erized where the first data represents the root and the other the suffixes. If it is desired to represent a series of dates such as 1976, 1978, 1983, 1985. Their representation could be 1976, 2, 5, 2 where all numbers after the first represent a difference from the value of the previous item.

Another method of representing differences is to use a bit map that indicates which of the fields from the previous key have been changed in the current key. The above sequence of dates are represented as:

1976, "0001"2, "0011"83, "0001"2

where "xxxx" represents the bit map for the item which follows.

The differencing scheme is a generalization of front compression. The process of compressing the front string is repeated for each maximal substring in the current record which matches a substring (in the same position) in a pattern record. The start and end signals for such a matched substring constitute the overhead for the scheme, the information unit on which differencing is performed can be the bit, the byte, the field, or logical information.

Differencing schemes seek to diminish the space required for data storage by not repeating that part of the data record that is already present in previous record. Most often, differencing is applied to sequential files where the pattern record is taken to be the previous record in the file, which may first have been sorted. If used with a direct access file the first record of the block should be stored intact since it is used for the direct access. This may be expensive when the compression ratio is a not small enough. Differencing is particularly good technique when used to compress indices.

For example, IBM uses a form of differencing (both front and rear differencing) when representing the stored indices in the sequence set of a VSAM file.

The use of the same pattern record for the whole file may not yield as good a compression as a scheme where the pattern used to compress each record is the record preceding it. But the latter choice is more expensive in encoding and decoding time. Indeed, whenever a record is to be read every record preceding it has to be decoded. It is clear that the operations of deletion and insertion are even costlier.

4.3.4 Statistical Encoding of Huffman

A statistical encoding is a transformation of the user's alphabet in order to convert each character of the alphabet into a code bit string whose length is inversely related to the frequency of the character in the text. If a text is expressed in terms of an alphabet of characters I=A1..An where each character occupies K bits then an efficient statistical encoding will assign a code Bi to each word, or combination of the Ai, such that

$$\sum_i Fi*Bi \leq K*N.$$

where Fi is the frequency of a word in the text, Bi is the length of the code Bi, and N is the number of characters in the text. Note that it could happen, when using statistical encoding, that a code assigned to an infrequent word requires more bits than the uncoded word.

An essential property of any statistical encoding scheme is that it be completely reversible in the sense that the original text may be retrieved from the encoded text in a finite (preferably linear) number of steps. Another desired

POOR COPY

quality is the prefix property which states that no code Bi shall be the prefix of another code Bj. This property assures both complete and unique reversibility and also that the decoder never has to back up and rescan any portion of text.

The Huffman coding scheme provides an effective simple statistical coding algorithm that has the prefix property. The code is space optimal in the instance that the probability of appearance of any letter is independent of the probability of appearance of any other letter. Huffman coding provides an easy and effective method of file compression without necessitating any inquiry into the semantics of file records, and the coding is suitable for files that are frequently updated. If the frequency of occurence of a letter is Fi, then the expected code length generated by the Huffman code is closely approximated by the entropy of the frequency table:

$$H=(Fi/\sum_i Fi)*Log_2 (\sum_i Fi/Fi)$$

where i ranges over all characters of the alphabet to be coded. In fact

H.LE. (expected code length).LE.H+1.

Thus, if a file is frequently updated, it is easy to compute new frequency statistics at the same time updates are performed. Huffman coding has the further advantage over differencing techniques that records can be decoded individually without need for some reference to a pattern record.

Employing Huffman's coding involves the construction of a binary tree as follow:

1) sort the messages into decreasing order of probability

2) combine the two messages of lowest probability to

form a new message which becomes the parent and replace them.

3) repeat (1) and (2) until all the messages have a common root.

4) for each original message, start at the root of the tree and go toward a message coding "0" each time one goes left and "1" when one goes right.

Huffman encoding is optimum in the sense that its cost in storage space is not greater than that of any other uniquely decipherable encoding.

Svank (35) described a simple technique which optimizes alphanumeric data storage. A reduction by a proportion of 0.3 or better in storage requirements can be realized. The basic philosophy of the approach is based on representing character pairs consisting of vowel/vowel and vowel/consonant by a combination byte of 8 bits. The potential saving in storage is a function of the percentage of such combination of pairs that occur in the data base. This proportion depends on the natural language used. In this context a vowel is defined as one of the characters A E I O U Y and blank. For example the sentence "this is a sentence to illustrate savings in storing alphanumerics data" will be coded as T/H/IS/ I/S/ A/ S/E/N/T/E/N/C/E /T/O /IL/L/US/T/R/AT/E /S/AV/IN/G/S/ I/ N/ S/T/OR/IN/G/ A/L/P/H/AN/UM/ER/IC/ D/AT/A. The coded sentence may be stored in 45 bytes instead of the 69 bytes needed for the uncoded sentence.

Since there are 7 vowels in the proposed model, the number of combinations necessary to represent all possible vowel/consonant and vowel/vowel pairs is 7*27 = 189. To improve the compression ratio special codes can also be assigned to represent some of the next frequent words of the natural language. Most of the techniques described above,

except the Huffman code, use character coding. But coding of alphanumeric data character by character is wasteful of storage space, and in dealing with a very large data base it is desirable to find a more compact representation of the information.

The method of Svank is almost the same as the one proposed by McCarthy (36) for automatic file compression. The principle used by McCarthy is to extend the basic alphabet of the data by adding to it a set of cords (where a cord is a string of two or more characters) chosen because of their high frequencies of occurence in the data; in such a technique difficulties arise because of overlap. For example, given the following string ABCDEFGABCDEFGABC with cords ABCD, CDEFG, and EFG the string could be "parsed" in either of the following ways (1) AB*AB** and (2) ***** where each asterisk marks a point from which a cord has been removed. The string displayed in (1) has a length of 7 characters and is clearly less satisfactory than that in (2) which has only 5 characters.

A similar method has also been proposed by Schieber and Thomas (37). In producing the code the requirements were that the procedures used for the choice of strings to be encoded be kept relatively simple, and that the encoding algorithm combine simplicity and speed, presumably by minimizing the amount of dictionary lookup required to encode and decode the selected string.

The basic technique used to compact the data file requires that the most frequently occuping digrams be replaced by single unused special character codes. Characters that are unallocated in the data base may be used to represent longer character strings. The most elementary form of substitution is the replacement of specific digrams. If these digrams can be selected on the basis of frequency then the compression ratio will be better than if selection is done independent of frequency. This requires a frequency count of

all digrams appearing in the data, and a subsequent ranking in order of decreasing frequency. Once the base character set is defined, and the diagrams eligible for replacement are selected, the algorithm can be applied to any string of text. Algorithms for both encoding and decoding are required.

In encoding the string to be encoded is examined from left to right. The initial character is examined to determine if it is the first of any encodable diagram. If it is not it is moved unchanged to the output area. If it is a possible candidate, the following character, is checked against a table to verify whether or not this character pair can be replaced (a list of digrams is given in (3.7)). If replacement can be effected the code representing the digram is moved to the output area. If not, the algorithm then moves on to treat the second character in precisely the same way as the first. The algorithm continues, character-by-character, until the entire string has been encoded.

Example

In the following example it is assumed that only three digrams are defined as codable: AB, BC, BE and DE. It is also assumed that the text to be encoded is the six character string ABCDEF. After encoding the coded string would appear as:

"AB" C "DE" F

Note that although BC was defined as an encodable digram, it did not combine in the example above because the digram AB was already encoded as a pair. The characters C and F do not combine, so they remain uncoded. Note also that if the digram AB had not been defined as codable, the resultant combination would have been different in this case.

A "BC" "DE" F

The decoding algorithm serves to expand a compressed string so that the record can be displayed or printed. As in the encoding routines, decoding of string goes from left to right. Bytes in the source string are examined one by one. If the code represents a single character, the print code for that character is moved to the output string. Decoding proceeds byte by byte as follows until the end of the string is reached.

1) load string length into counter.

2) set pointer to first byte in record.

3) test character; if the code represents a single character then point the next source byte and retest.

4) if the code represents a digram: move all bytes (if any) up to the coded digram, and move in the digram

5) increase the length value by one, point the next source byte and continue with step 3.

There might be good reason to extend the technique to the encoding of longer strings provided a significant higher compaction ratio could be achieved without undue increase in processing time. One could consider trigrams, quadrigrams, and up to n-grams. The English word "THE", for example, may occur often enough in the data to make it worth coding.

Heaps (38) introduced a non alphabetic compression code based on the frequency of occurence of words in a document data base. The important advantages of the coding scheme over the alphabetic scheme are that the codes have smaller average length and may be decoded uniquely to reconstruct the uncoded term. Addition and deletion of terms is easy, because the code is independent of the previous entry, or of

general patterns, furthermore the code may be constructed by purely automatic means.

The scheme may be described as follows: "let N1,..., Nm be a sequence of positive integers. The most frequent 2** (N1-1) terms are assigned codes of N1 bits in which the left hand bit is set to 0, the next most frequent 2** (N2-2) terms are assigned codes of N2 bits in which the first bit is set to 1 and the (N1+1)th bit is set to 0. Similarly, the next most frequent 2**(N3-3) terms are assigned codes of N3 bits in which the first and the (N1+1)th bit is set to 1 and the (N2+1)th bit is set to 0".

Some systems that involve a large data base generate special problems that are not found in smaller systems. Library retrieval systems are in this group. That is the reason why Reid and Heaps (39) introduced a new method for compression in a library environment and compared it with previous well-known methods in word compression, Schuegraph and Heaps (40) analysed the minimum space (MS) algorithm and the longest fragment first (LFF) algorithm that are special algorithms for library data base compression by use of word fragments.

4.4 Practical Application

When taking the decision on whether to use compression for an application, one has to consider the following items

1) compression is suitable primarily for mass storage files and communications. Compression on tape files is seldom practical because the gains in storage cost, reels of tape, is minor and the tradeoff between channel time saved and CPU time spent is marginal and often unfavorable.

2) compression is especially suitable for large files, where savings in storage costs can be substantial.

3) compression tends to be more feasible on fast computers or on systems which are I/O bound. Medium speed or slower computers or systems which are primarily compute bound are limited in the amount and type of compression that is feasible.

4) compression is more suitable for files which are accessed randomly. The cost of compression is minor compared to the cost of a random access to mass storage and increases the run time in random programs by a very small percentage. On sequential runs, however, where I/O is blocked and each record in the file must be decompressed and compressed, the effect of compression of run time can be drastic.

5) feasibility depends to a great degree on the amount of redundancy in the file being considered. Files with many zeros and blanks are, of course, good candidates.

4.5 Conclusion

The importance of compression for mass storage is related to the cost of mass storage devices. The per-character cost of mass storage has had a downward trend, but due to increased use, the amount spent for mass storage has gone up. New developments, such as high density recording devices which enable more data to be stored on current mass storage devices, laser memories or other breakthroughs may further reduce the cost of mass storage and hence reduce the importance of compression; However, it is unlikely that the cost of mass storage will become low enough to eliminate the need for compression altogether.

In communications, compression is likely to increase in importance and find a larger place in remote networks for business activities.

CHAPTER 5

FILE ORGANIZATIONS

5.1 Basic File Organization

Since the advent of computers to treat large data bases much attention has been devoted to the subject of generalized data base management systems and file organization. The CODASYL Data Base Task Group has been a leader in those areas. This has been in parallel with the emergence of computer-based management information systems, MIS, in today's large and complex organizations.

Various generalized data management systems are in use in many organizations. Among these packages are MARK IV, ASI-ST, IDS etc, but the kind of file organization used has a very important effect on the performance and associated costs of such systems. These costs usually include total storage costs and average access time per query, which are two very important quantifiable performance measures for judging a data base organization, and are of concern whether batch or on-line systems are involved. In some situations important performance parameters may depend heavily on the structure of the file management system. One such parameter is the average time needed to answer a query in an on-line environment. Some other factors also influence the performance of a file organization, among them are the costs of update and the highly unquantifiable cost of programming complexity, reprogramming and maintenance.

The basic file organizations or primary indexing methods supported by present day computing systems are the well-known sequential, random, and indexed sequential organizations.

5.1.1 Sequential Organization

The best known data organization is the sequential organization, wherein records are stored in positions relative to other records according to a specified sequence. To order the records in a sequence one common attribute of the records is chosen. When it is a data item within the record the attribute selected to order the records in the file is called a key. By selecting a different key for a file, and sorting on the basis of that key, the sequence of the records in the file may be changed. Records may be stored without keys; this occurs when records are stored in order of their arrival into the system, each record being positioned sequentially following the preceding record. In both cases the logical order of the records in the file and the physical order of records on the recording medium is the same. A sequentially organized file may be composed of records of different types, but the records are grouped into a file because they have a common functional purpose.

The advantage of sequential organization is fast access per relationship during retrieval. If the access mechanism is positioned to retrieve a particular record, it can rapidly access the next record in the storage structure according to the relationship which was established when the data was stored. If the records are stored in sequence on a direct access medium or in core memory, a binary search is possible. The advantage of being able to rapidly access the next record becomes a disadvantage when a file is to be searched until a record having a particular key value is encountered. Here the first record is examined; if the key is not correct, the next record is examined and the process continued until the correct record is found.

If the new record is shorter or longer than the original record, adjacent records in the file may be destroyed or

become inaccessible to the hardware when the record is rewritten. For these reasons a file having sequential organization is usually updated by copying records from one file to another, making updates to the records as needed. It becomes expensive to update one record in a sequentially organized file; usually updating occurs only when a number of records are to be altered.

It is also difficult to insert new records into, or remove old records from, a sequentially organized file. The insertion process requires that previously stored records be pushed apart to make room for a new record, resulting in the copying of the entire file. When removing records existing records are pushed together.

Another weakness of sequential organization occurs in retrieval of records using more than one key. This is accomplished by storing the data in several files so that the first file is seached if the first key is known and the second file is searched if the second key is known, and so forth, this multiplication, however, uses more storage space and increases maintenance cost since a record to be updated must be accessed and changed in all files.

5.1.2 Random Organization

In random file organization, records are stored and retrieved on the basis of predictable relationships between the key of the record and the direct address of the location where the record is stored. The address is used when the record is retrieved. The relation between the key and the address of the record is very important for dealing with randomly organized data, and this relation can be built by three methods: direct address, dictionary look-up, and calculation.

POOR COPY

Direct Access: In the simplest form the direct access is known to the programmer and is supplied at storage and retrieval times. The hardware then uses the address to access the record on the storage medium.

Dictionary Look-Up: When this method is used, both the records key and its direct address are stored in the dictionary. When a record is stored or retrieved, the key is found in the dictionary and the corresponding direct access address is used. The use of a dictionary insures that each record has a unique address. However, the dictionary must be large enough to include all potential direct addresses and it may therefore occupy as much space as the data itself. Also the step-by-step sequential search of a dictionary may offset the gains offered by having unique record addresses. If the dictionaries are very large they are often accessed with a binary search to reduce search time.

The Calculation method involves converting the key of record into a direct address which is not necessarily unique. This method is similar to the hash coding scheme.

The advantage of using a random organization is that, in the absence of collisions, any record can be retrieved by a single access. Other records in the file do not have to be accessed and their keys examined as in the sequential method when trying to retrieve a particular record. Individual records can be stored, retrieved, and updated without affecting other records on the storage media. However all records are generally of a uniform length. This requirement is sometimes imposed by the way hardware addresses reference the storage media and also permits easier handling of overflow records.

Random organizations are used most often in conjunction with direct access storage media, and the direct address of a record corresponds to a hardware address on the storage device.

Although random organization does allow for rapid access of a particular record with a known key it is not suited for rapid access to a number of records. Other difficulties are (1) handling the overflow problem when the calculation is used to obtain the direct address, and (2) manipulating large and unwieldy dictionaries when dictionary look-up is used.

5.1.3 Indexed Sequential File

An indexed sequential file organization is shown in Fig. 2.

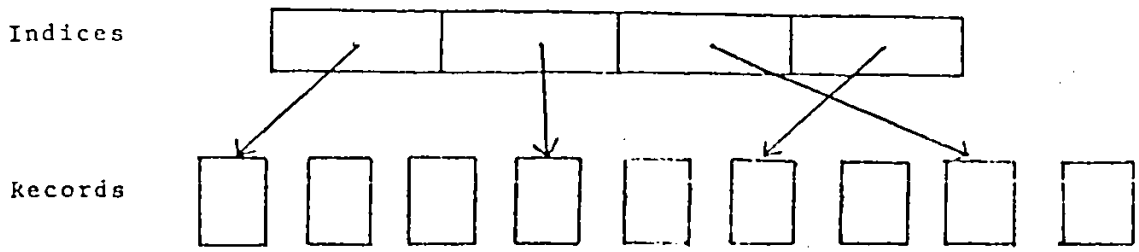


Fig. 2

This file has the property that records can be accessed either by the use of indices or in sequential fashion. The indices are stored sequentially, and each index contains address of a record in the file. If a record is to be retrieved by its index, the indices are searched until the desired key is found, and the pointer from that index to the record is used as a direct address to retrieve the record. The records are also stored on the storage media in a sequential fashion, permitting the records to be accessed in a sequential order. Indices in the indexed sequential file are often oriented to the characteristics of the storage media. In some systems there is a cylinder index as well as

a track index for each data item stored in the file.

5.2 Secondary Index Organization

The manner in which the above organizations work may vary from one computer system to another. These variations are visible at the level of higher languages such as Cobol, but are perhaps more noticeable at lower levels and at the data management levels of operating systems. Some of these differences, many due to inherent differences in operating systems, have caused much of the often costly lack of data base transferability, reprogramming efforts, etc.

Because of the great mismatch between these basic facilities for data management and the heavy information retrieval demands placed by modern users, various important file organization concepts and schemes termed secondary index organizations have emerged in the hope of allienating the mismatch. The main ones are: list and multilist structures, partially inverted and inverted files, chained and hierarchical files.

5.2.1 List Structure

The basic concept of a list is that pointers are used to divorce the logical organization from the physical organization. In a sequential organization the next logical record is also the next physical record. However, by including with each record a pointer to the next logical record, the logical and physical arrangement may be made completely different. A pointer may be anything which will allow the accessing mechanism to locate a record. If the records are stored on a direct access media it is the direct address of the record; if the records are in core memory it is the core address of the record. The logically sequential organization

is obtained by using pointers. Initially, there is a pointer to the first record, and the last record in the list is designated by a special symbol.

Since any data item in a record may be treated as a key, many lists can share a single record. By allowing records to be on multiple lists, duplication is avoided. If a record is to be updated, it can be updated in all lists automatically by updating it in one list. Records can be placed anywhere within a list by changing the pointer of the record preceding the new record and inserting a new pointer in the record just inserted. The removal of a record from a list requires changing the pointer in the preceding record to point to the record in initially accessed from the preceding record. However, if a record is a member of several lists it becomes more difficult to find the preceding records of all the lists and extra pointers must be stored so that the preceding as well as the next record of the list may be found.

With large files the lists themselves tend to be long, and extended searches may be required if the list length is not controlled. Where the list length is not restricted, there is only one starting point for each list. Where the list length is restricted, the effect is to create sublists, each of which has its own starting point. This reduces the search times at the expense of mainteuing an expanding index of starting points.

5.2.2 Inverted File

The inverted file is usually described in terms of accession numbers (pointers to records of addresses of records) and index terms (key values or access keys). A unique accession number is assigned to each record in the file. Associated with the file is a file of index terms. Attached to each index

term is a list of accession numbers for the records that are characterized by the index term. The basic structure is shown in Fig. 3.

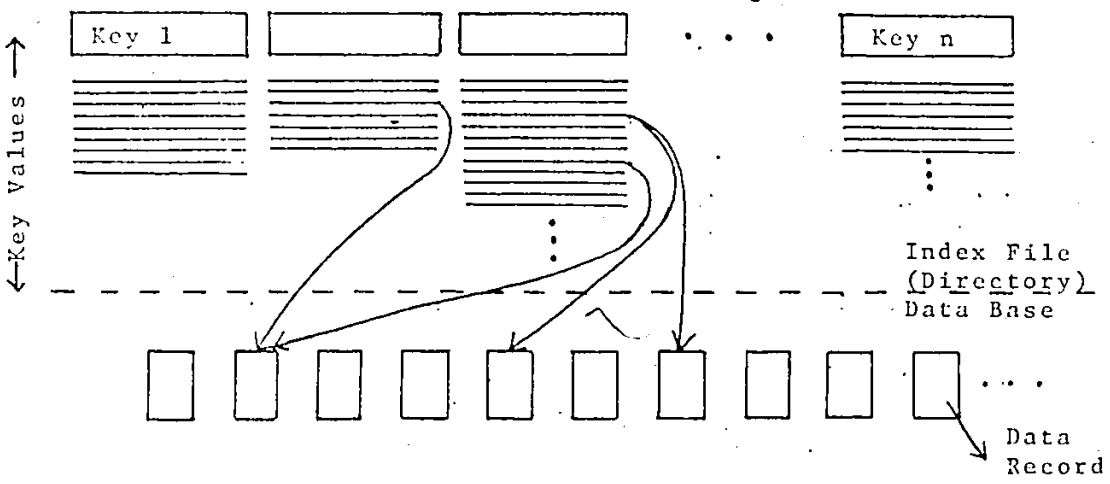


Fig. 3

The inverted list organization makes available every data item as a key. Such an organization requires a dictionary of all data values in the system containing the addresses of all locations where those values occur. The dictionary can be as large as, or larger than, the data itself. The virtue of such a system is that it allows access to all data with equal ease. Consequently, it is more suitable for situations where the data retrieval requirements are less predictable, for example decision-making and planning functions, than it is for specific processing functions. In order to answer a query the index file can be quickly searched for the access keys specified in the query. Although the inverted list approach lends itself to easy retrieval, storing and updating data is more difficult because of the maintenance of the large dictionaries. It is best to combine an inverted list

organization with either a sequential or random organization. If records are inverted on only one or two keys the dictionaries become smaller and it is still possible to access all records in the file. The extent to which key values of the file appear in the index defines the so-called degree of file inversion.

Inverted file searches are usually two-step processes. First, the matching documents accession numbers are retrieved from the inverted file, and then the documentary information is retrieved from a primary file using the accession number as addresses.

5.2.3 Multilist File

In the multilist structure each access key represents a linked "list" of data records. The index directory contains the access keys and a pointer to the first in the list of records. Rather than storing the pointers to data records separately as in the inverted list, the multilist stores the pointers within the data records. While the storage requirements are approximately the same, the average access time can vary considerably.

The records are stored in fixed-size blocks called cells or pages. They are addressed by a page number and a record number within the page: the record identifier (X,Y) references the Yth record in the page X. The grouping of records into pages is an important concept used in storing list structure on direct-access media. Rather than using a record's direct address as a value in a pointer, a page number and record number are specified. The page number is used as a direct address to retrieve the page, and the record number is used to locate the record within the page. In this way a group of records can be retrieved at one time and the access time per record is reduced. Saving is achieved only if a number of

records in the page are desired. It is inefficient to retrieve an entire page to process only one record. In using a multilist file it is therefore preferable to store related records within a common page. To eliminate unnecessary accessing of pages, it may be desirable to limit list length so that all the records on a list fit in a single page.

In summary, a multilist file consists of a sequentially organized index with each index entry pointing to a list of records. Records in the file are blocked into logical units called pages. An entire page is accessed when a record is stored, updated or retrieved. A multilist structure is diagrammed below in Fig. 4.

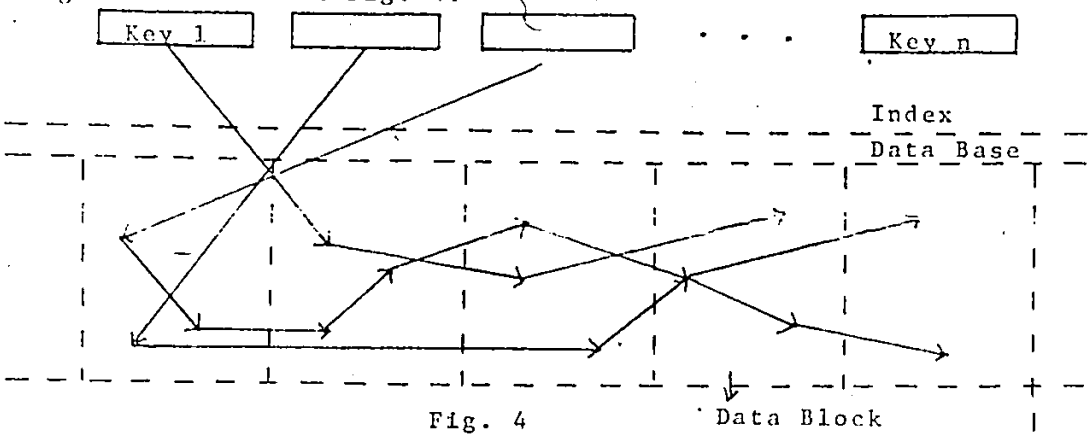


Fig. 4

5.2.4 Tree (Multi Level) Organization

A tree data organization is one in which several levels of indices are used. The index on the first level points to a collection of indices on a second level. One of these second-level indices is used to find a collection of indices on a third level, and so forth, until the actual data value has been found at the bottom of the particular branch on this tree. At this lowest level of the tree are the data base records.

There are usually three pointers associated with each

node ; one points to a set of key values on the next lower level which are in those records having this key; another to the next key value in the same level; and the third to the key value on the higher level that is the predecessor of the filial set in which this key is located.

Physically the storage structure contains two types of blocks: index and data. The data blocks contain the original records without key values. The index blocks contain the key values and pointers. It is assumed that the file will be accessed via the access keys. Sufficient pointers are supplied for reconstructing the original record from any point in the tree. The tree organization is conveniently used in maintaining large dictionaries. Portions of the dictionary can be added, deleted, or changed without disturbing the rest of the organization. Fig. 5 is an example of a tree organization.

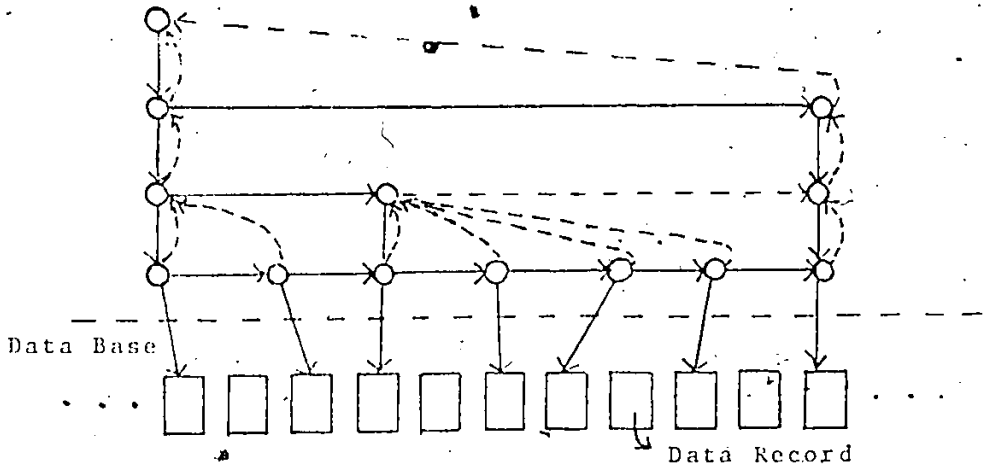


Fig. 5

5.3 File Organization Performance

Once a file organization has been chosen it is desirable to estimate its effectiveness. A very important question is: what are the critical measures and parameters that can be examined and controlled to optimize the performance of various

POOR COPY

file structures? Before answering this question we have to agree on the meaning of optimum or at least "best and appropriate" file structure? What is "best" or even "appropriate" depends on the particular users and environments. It is proposed by Cardenas (41) that the best structure is one which minimizes the following cost function (CF) and at the same time satisfies design constraints:

cost/unit period = (cost per track/unit period)*

(total storage required in tracks) + [(cost/second)*

(access time in seconds per average query) *

(number of queries, per unit period)] +

[an update cost function] + [other highly

unquantifiable cost functions].

(unit period could be a day or a week or a month, etc).

Design constraints such as that "query response time must not be more than three seconds for most queries" add a further consideration for determination of what is the best file organization.

Total storage costs and average time to answer an average query in (CF) can be quantified. The update cost function is more difficult to define and quantify because of many multiple side effects. The following is an example: under OS/360 when records are added to a data base structure, such as an inverted file which is based on indexed sequential methods, the new records (or original records forced to be shifted by the insertion of the new records) are placed on so-called overflow

areas. Records in overflow areas take longer to access than records in the original prime area. When records are deleted under OS/360 the space freed is not necessarily made available for other new records or shifted records. The result is that file performance can be so degraded that the whole data base has to be "purged", that is dumped out, and then read in again so that it occupies only prime area.

The fourth element in (CF) may be termed "other highly unquantifiable cost functions" and it includes the costs of initial programming, reprogramming, software maintenance, documentation and various others tasks. Even if they are difficult to evaluate, they have to be taken into consideration within the evaluation of the total performance.

The access time, storage cost, and update time performance of a file organization depends on: (1) data base characteristics, (2) user requirements, (3) storage device specifications and (4) file structure characteristics and programming techniques used to implement the structure. In fact the particular approach and programming techniques used to implement each file structure may greatly affect its storage, access, and update time characteristics. Programming languages, methods of mapping onto storage, and particular programming tricks may have a large effect. The specific basic file organization used is also a factor. For example, it is noted that there are three versions of the random access file organization supported in PL/1: Regional-1; Regional-2; Regional-3 (42). Each version uses different methods to manage storage and access records, although from the point of view of the PL/1 user these differences are essentially transparent. A specific case is the following difference between Regional-3 and the other two versions as implemented in IBM OS/360. When a record is deleted from a Regional-3 file the space vacated is not made available

for future use. In other words there is no garbage collection, whereas in the two others it is. So in Regional-3 the user still pays in storage for "deleted" records.

Another, and perhaps the most important factor affecting total file structure performance, is the manner in which the indexes or dictionaries and lists are managed and placed in storage. In highly-inverted files with many different key values the directory becomes another file problem in itself, possibly of the same magnitude as the data base itself. When the lists are long it will matter just where and how the pointers are managed. When a key value of a record is inverted, is the key in the record deleted in order to conserve total storage requirements or is the key left in the record to avoid programming problems of compacting records at the expense of storage? Several cases of this nature could be cited to point out their effect on file structure performance aside from the effects of storage device specifications (e.g. time to position disk arm and access the desired address) and of data base and query characteristics.

5.4 Methodology of Evaluation

One of the methodologies used to compare or select an appropriate data base structure is: 1) obtain measures of the relevant characteristics of the actual data base or of a representative sample of it, 2) estimate the requirements of the user, 3) take into account device-related specifications supplied by the manufacturer and 4) estimate by simulation the minimum average access time to answer an average query for various alternative file organizations. On this basis, an appropriate file structure may be selected. In any case, if the data base is frequently updated, the decision should also take this into account.

POOR COPY

The specific data base statistics considered most pertinent to be measured are:

- number of records.
- minimum record length.
- maximum record length.
- average record length.
- average key-value length.
- number of unique key-values for each key.
- minimum key-value length of each key.
- maximum key-value length for each key.
- average key-value length for each key.

The relevant user requirements may include the following which is not an exhaustive list:

- frequency of query, frequency of update.
- mode of retrieval, mode of update - batch or on-line.
- type of update: insert records, delete records, modify records.
- complexity of the queries.
- average number of records retrieved for a query (an estimate)

- file structure desired.
- number of access key names used to access records.

The device specifications that should be taken into account are:

- block length.
- average time to access a track.
- average time to compare an access key.
- average time to intersect two blocks of access keys.
- cost per track per unit period.
- machine cost per second.

The simulation model of each file structure depends very much on just how the structure is implemented, consequently the implemented simulator will be one of various possible implementations (the one considered "most appropriate") of the file structure. That is the reason why the list of parameters given above may be a function of not only the type of file structure, but also, to some extent, of the particular implementation approach.

In conclusion, the performance of all file organizations, both primary and secondary, depends on: (1) data base characteristics, (2) user requirements, (3) device related specifications, and (4) the particular approach and programming languages and techniques used to implement the file organization chosen. The last factor is often underestimated. The kinds of programming approaches and techniques used, and particularly the manner in which the secondary indexes or dictionaries and associated lists are managed, may have underestimated impacts.

CHAPTER 6

PROPOSED HASH ADDRESSING AND FILE ORGANIZATION

6.1 Choice of a Hashing Scheme

One of our basic objectives is the production of codes of length 6 * N, N=1..3, whose 5 lower bits will be determined by a hashing function. Accordingly, special attention has been given to this hashing function. The hashing method used is the "quadratic quotient method" modified for a full table search. To allow use of bit code (1 for a flag and five for a table-address) the table is constrained to have a maximum size of 32=2**5. The hashing scheme should allow a full table search in a table whose size is a power of 2.

However, it is known that when the size of the table is a power of 2 the period of the hashing function is shorter than when the table size is a prime number (20). Furthermore execution of the operation of modulo on a CDC computer when the divisor is a power of 2 produces too many identical home addresses due to the fact that literals are padded with blank (55B) to fill a word and even if the non-blank character is chosen as a key the corresponding integer will be padded with '00B.

In order to avoid this, we introduce the concept of virtual size for a table, and choose the virtual size as equal to the smallest prime number greater than the real size and of the form 4*X + 3 where X is an integer. In the particular case illustrated the real size is 32 and the virtual size 43. When the home address falls between the real size and the virtual size it is treated as a collision occuring in the table of real size equal to the assumed virtual size.

The hashing functions operate on the first c1 and last c2 characters of each descriptor, where a descriptor may consist of any string of characters chosen from the sets A to Z and 0 to 9. The values of c1 and c2 are chosen to depend on descriptor length as follows:

descriptor length	c1 and c2
1	1
2	1
3	2
4	3
>4	4

The algorithm used to compute the home address is as follows:

1) place the character string of c1, or c2, characters left justified in one 60 bit word of memory.

2) shift the string 12 bits to the right in order to form a new integer.

3) regard the character string as a binary integer, divide it by the table size, and save the remainder.

4) use the remainder as the hash address.

If a collision occurs the search function proposed by Collin Day (this function is described in chapter 3) is used (26). It guarantees a full table search.

6.2 Experimental Test of Hashing Scheme

For study of the relative merits of the hashing scheme it has been compared to the regular method for 13 different sets of data. These 13 sets of data were chosen from the Kucera and Francis table, the Chemical Tapes, and general English or French words. One of the set consisted of the most frequent words proposed by Kucera and Francis in their table. The 13 lists of terms used during the experiments are given in Appendix A. The relative merits of the 'virtual size' concept are studied in the following four cases.

1) use of the virtual size concept with a load factor 1 (32/32).

2) virtual size concept and load factor .72 (23/32).

3) virtual size concept and load factor .75 (24/32).

4) the concept of virtual size is not used and the load factor .75 (32/43).

To study the influence of load factor and concept of virtual size on the efficiency of the method, tests have been made on 13 different sets of data. In all except one instance the load factor alpha was greater than 75 percent so that up to 75 percent of the table capacity was used; in the remaining instance the full table was used so that the load factor was 100%. The results are summarized in tables 1 to 4. They are valid only for the one dimensional hashing in the first level, where the entire word is used.

It may be remarked that even if the method degenerates, because of the table size chosen, the mean number of required searches is found to be less than or equal to the one predicted

by Maurer for random probing (2.56) or linear probing (5.50), in the case that the load factor is equal to 0.9.

6.3 Some Observations on the Results

a) table 1 contains the average number of accesses to store an element:

the number of accesses required by the proposed scheme is less than the one observed by Lum (table 8) and the theoretical number of accesses predicted by Peterson. We can see that we obtain different results in the two cases where alpha = 75 percent, this shows that the efficiency of the hash method is not only a function of the load factor, but depends also on the size of the table and the vocabulary of the data base.

b) table 2 contains the total number of collisions:

the results are comparable to those obtained by Lum except when the table is full. But even in this case the efficiency of the scheme does not suffer because the average access time is still lower than the one observed by Lum. This may be attributed to the fact that Lum used the linear search for handling collision. While in the present investigation use of the quadratic search allows the average number of probes to be smaller than the one obtained by Lum.

c) table 3 contains the longest sequence of collisions:

it explains why the average access time is low. This is because there is a very low average of 'longest sequence of collisions'.

d) table 4 contains the time evaluation:

the loading factor does not appear to have a significant effect on the time efficiency. Second is the time unit used here.

Conclusion

Examination of the results summarized in tables 1-4 suggests that hashing through use of the 'virtualsize' concept does not deteriorate, and sometimes improves, the traditional method of through use of a prime number for the table size.

6.4 Predicted Performance of the Double Hashing Scheme

The above results have been obtained through use of a single hashing function to store an element in a one-dimensionnal array. But in a document retrieval system it may be desired to allow left and/or right truncation, and this may complicate the retrieval process. To facilitate use of query terms with truncation the principle of double hashing has been introduced. By storing an element in a table INDICT2(32,32) at an address based on H1(c1) and H2(c2), where c1 and c2 are derived from the original word, there is a tendency to formation of clusters associated with prefixes or suffixes. For example words like computer, computers, will be assigned to the same row, and a query that contains "comp*" will be answered by scanning the row selected by evaluating H1(comp). A similar process is performed to process queries containing "*tion", because most of the words terminating by tion will be stored on the column designated by H2(tion). It is of interest to consider the distribution of the new function (H1oH2).

Distribution of (H1oH2)

The probability distribution of an element in INDICT2(32,32)

can be computed as follows:

$$K = (I-1)*32 + J$$

$$P(K=k) = \sum_i p_i * P(K=k \mid I=i)$$

$$= \sum_i p * P(J=k-32*(i-1)$$

But

$$P(J=k-32*(i-1) = \frac{1_{[1 \le k-32*(i-1) \le 32]}}{N}$$

where 1 is a function = 1 if $1 \le k-32*(i-1) \le 32$
0 otherwise

and N = Total of number of Keys present in the table

If we assume that all the $P_i = 1/N$ then

$$P(K=k) = \sum_i 1/N \cdot \frac{1_{[1 \le k-32*(i-1) \le 32]}}{N}$$

$$= \frac{1}{N^2} \sum_i 1_{[1 \le k-32*(i-1) \le 32]}$$

Now $1 \le k-32*(i-1) \le 32$ implies $\frac{k}{32} \le i \le \frac{k+31}{32}$

and so if we define and integer function M_k such that

$$M_k = \text{Int}\left[\frac{k+31}{32}\right] - \text{INT}\left[\frac{k}{32}\right] \qquad k > 0$$

then $P(K=k) = \frac{M_k}{N^2} = \frac{1}{N^2} \quad \forall k$

so we can conclude that:

the distribution of H1∘H2 is as uniform as the distribution of H(key) = 1/1024. The result is that as H1 and H2 generate uniform addresses, the use of double hashing does not decrease the efficiency of the resulting function. But we observe that it increases the number of collisions. The average number of accesses for an element in INDICT2 is 35.5 in comparaison to 2.41 for a one dimensional array INDICT1 addressed by a single hashing function H1.

This may be a consequence of the way collisions are handled in INDICT2. First we determine a row by computing H1(c1) and then we try to place the term in the column of that row. This position is determined by H2(c2). If a collision occurs we solve it by means of the quadratic quotient method, which is used to find a free column. When the row is full, we select another one by use of the linear search method. As this method does not solve secondary clustering the sequence of collision is longer.

This algorithm is used if we have more than 20 percent of empty positions in the chosen INDICT2, otherwise if the row determined by H1(c1) is full we try to put the term in the same row but at a lower level in the hierarchy of dictionaries.

Probability of Collision in a Double-Hashing Scheme

Suppose the Nth descriptor is to be inserted as Xn. The probability that this particular element is assigned to the Kth position is:

$$\forall n \qquad P(X_n = k) = p_k = \frac{1}{N^2}$$

POOR COPY]

The probability that a collision occurs with the first inserted element is $P(X_1 = \text{Collision}) = 0$

If we have a collision when the second element is inserted this means that X_1 and X_2 should be placed in the same Kth position $\forall$ k.

$$P(X_2 = \text{collision}) \sum_{K=1}^{1024} p(X_1 = k) * P(X_2 = k)$$

$$= \sum_{K=1}^{1024} p_k^2$$

$$P(X_3 = \text{collision}) = \sum_{k_1 1,\ k_2}^{1024} P(X_1 = k_1 \text{ and } X_2 = k_2) * P(X_3 = k_1 \text{ or } k_2)$$

$$k_1 \neq k_2$$

$$= \sum_{\substack{1 \\ k_1,\ k_2 \\ k_1 \neq k_2}}^{1024} (p_{k_1} * p_{k_2}) * (p_{k_1} + p_{k_2})$$

So by induction we can assume that

$$P(X_n = \text{collision}) = \sum_{\substack{1 \\ k_1,\ k_2,\ \ldots k_{n-1} \\ k_i \neq k_{i+1} \\ 1 \leq i \leq n-2}}^{1024} (p_{k_1} * p_{k_2} * \ldots * p_{k_{n-1}}) * (p_{k_1} + p_{k_2} \ldots + p_k$$

POOR COPY

Practically the results obtained by the double hashing scheme were satisfactory and are summarized in table 9 and table 10. Best results have been obtained using H_1 (C_1) because the number of synonyms in both lists is not the same; 15 percent for C1 against 25 percent for C2. Furthermore in C1 11 percent of synonyms came from sublists of two different words (ex. I, IN), while in C2 this percentage is only 6 percent (ex. AS, IS).

6.5 Proposed File Organization Model

The proposed file organization is designed to allow implementation of the compression technique described by Heaps (38) and also to minimize the number of disk accesses required during coding and decoding. To achieve this aim a file organization similar to a paged-segmented memory is chosen to provide hierarchical storage where only the first level is in central memory. The access to each level is made by means of a hashing function, and the proportion of data base text that appears in each level may be predicted by use of Zipf's law. Each level of storage contains:

1) a dictionary of the descriptors.

2) an inverted file of document numbers of the documents that contain such descriptors.

Associated with the dictionaries and inverted files are inverted tables containing pointers. The code associated with each descriptor will be stored in a compressed data base. The file organization is intended to be independant of the computer system (hardware and software) in order to permit a high degree of transportability.

Organization of the First Level.

A set of 32 very frequent descriptors are stored in Dict1

which is in core. These 32 descriptors depend on the vocabulary of the data base, and they are padded with right blanks to occupy L32 characters where:

L32=maximum length in characters of each of the 32 descriptors.

If the 32 descriptors are chosen as the most frequent ones it may be found that L32 is considerably greater in value than their average length; for example it might be the length of only one of them. In such an instance it proves convenient to select the 32 descriptors as the 32 most frequent of less than a given length. Thus in selection of descriptors from the Kucera Francis table the descriptor 'which' of length '5' has been displaced by the less frequent descriptor 'you' in order to allow the value of L32 to be significantly reduced from 5 to 4. This allows a saving of 182 bits which represents 3 percent of the total storage required for DICT1. In some data bases, such as the Chemical Titles Tape, such a policy may not be necessary because the distribution of the length of the most frequent terms is more uniform.

The 32 most frequent descriptors are stored in DICT1 by use of a hashing function instead of alphabetical order which would allow a binary search. Two simulation programs were used to test the two methods, and the results show the hashing scheme far superior to the binary search. The data used during the simulation were taken from the K.F tapes and are listed below in the pattern (descriptor-frequency).

---

the = 69971 , of = 36411

POOR COPY

and	= 28852	to	= 26149
a	= 23237	in	= 21341
that	= 10595	is	= 10099
was	= 9816	he	= 9543
for	= 9489	it	= 8756
with	= 7289	as	= 7250
his	= 6997	on	= 6742
be	= 6377	at	= 5378
by	= 5305	I	= 5173
this	= 5146	had	= 5133
not	= 4609	are	= 4393
but	= 4381	from	= 4369
or	= 4207	have	= 3961
an	= 3747	they	= 3618
which	= 3562	on	= 3292

The results can be summarized as follows:

	Binary search	hashing	hashing (v.s)
(1) Long search	Ln(32)=5	6	12
(2) mean numb	39,723.68	11,651.72	10,701.87
exec time	320 sec	47.36sec	46.61sec

(1) long search = maximum number of probes in order to retrieve a descriptor.

(2) mean numb = mean number of probes needed to retrieve a descriptor during the simulation process.

Proportion of Data Base Text Represented in DICT1

Suppose there are D distinct terms in the data base and they are ranked in order of descending frequency of occurence. Then according to Zipf's law the probability with which a term of rank R occurs in the data base is given by

$$PR = A/R$$

where A is a constant which must satisfy the relation.

$$AT(1+(1/2)+(1/3)+(1/4)+\ldots+(1/D)) = T$$

where T = total number of words in the data base.

Thus

$$A = 1/(1+(1/2)+\ldots+(1/D))$$

$$= 1/(LN(D) + G)$$

where G = 0.5772 is Euler's constant.

Suppose DICT1 contains D1=32 terms of the data base. Then the proportion of data base terms contained in DICT1 is

$$P1 = A(1+(1/2)+...+(1/D1) - A(LN(D1) + G)$$

$$= (LN(D1) + G)/(LN(D) + G)$$

thus

$$P1 = (LN(32) + 0.5772)/(LN(D) + 0.5772)$$

For certain values of D the values of P1 are as tabulated below

D	P1
12,000	0.40
50,000	0.36
200,000	0.32

Storage Required for DICT1

DICT1 contains the 32 most frequent descriptors. The average length of such descriptors in the Kucera Francis list is 2.7 characters, and so DICT1 could be stored in 32*2.7 = 86.4 bytes.

Since DICT1 contains the most frequent words it will be

accessed very often during the decoding. One way to speed up the decoding process is to store DICT1 in an associative memory which is a memory device that stores in each cell information of the form (K,E), where K is a "key" and "E" is a pointer to the location where the information related to K is stored. K is one of the 32 most frequent descriptors of the data. The memory is accessed by using K to determine the location of a cell. If the cell contains (K,E) for some "E" then the memory returns "E", otherwise it signals "not found". The search of all the memory cells is done simultaneously so that access is rapid. The entry "E" will also serve as an index in ININV1. (see fig. 6)

The main advantage of an associative memory is that stored data item can be retrieved by their content or part of their content. From the hardware point of view, in order to retrieve stored data items by their content or part of it, one must be able to access the memory words by matching their content or part of their content with the given search key words, instead of by an address as in a location-addressed memory. The basic memory element of the associative memory is called the bit-cell. It has the property that one-bit information can be written in, read out and compared to the interrogating information.

The search operations, which consist of masking and comparison, are executed in a way that depends on the organization of the associative memory. The search-key word can be compared to all the words in the memory through the interrogating bit drives and the comparison logic circuitry. The possibility of matching multiple words to a search-key word requires that an associative memory have some method of tagging all the matched words. The tag function and matched-word indication are performed by the so-called tag networks. A complete survey of associative processor architecture is given by YAU S.S and FUNG H.S (44).

INV1 contains the list of documents in which each descriptor occurs. In each element of that list there will be stored an integer for the document number, and a "5-bit" code that indicates in which field of the compressed data base this descriptor appears. The code is assigned in the following manner: if a descriptor occurs in a field a corresponding bit is set to "1" within a five bit flag according to the following rules:

"10000" to indicate a title field (tit)
"01000" to indicate a abstract field (abs)
"00100" to indicate a author field (aut)
"00010" to indicate a publication field (pub)
"00001" to indicate a keyword field (key)

According to the Zipf's law the number of documents that contain the Ith descriptor is

$$T/[(LN(D) + G)*1)] \qquad (1)$$

where T = total number of words in the data base.

The average difference between successive document numbers in the list for the Ith descriptor is

$$DELTA(I) = (N/T)*(LN(D)+G)*I \qquad (2)$$

where N is the total number of documents, thus

$$DELTA(32) = (N/T)*(LN(D) + G)*32.$$

thus averaged over all 32 lists, the average difference between successive document numbers is:

$$DELTAV1=\left(\sum_{i=1}^{32} \frac{T}{(Ln(D)+G)*i} * \frac{N}{T} * (Ln(D)+G)*i\right)/C$$

where $C = \sum_{i=1}^{32} \frac{T}{(Ln(D)+G)*i}$

$$DELTAV1 = \frac{32*N}{T} * \frac{(Ln(32) + G)}{(Ln(D) + G)}$$

The dependence of DELTAV1 and DELTA(32) on D and I is summarized in the Table below:

D	T=10*N	DELTAV1	DELTA(32)
12,000	100,000	8	32
50,000	1,000.000	9	36
200,000	10,000,000	10	41

If the document numbers in INV1 are each stored in 20 bits, which allows for occurence of one million documents, then the total storage required for INV1 is 20*P1*T bits, where P1 is the proportion of text that appears in DICT1. Alternatively if difference coding, or the equivalent, of 6 bits for each document number could be used the space required for storage of INV1 would be reduced to:

6*P1*T BITS = P1*T characters.

However, even if most differences between successive documents may be stored in 6 bits there are likely to be some differences that are too large. Any such differences may be stored in the form of N+1 zeros followed by R where:

N = Difference DIV 63

and

R = Difference MOD 63

the distinction between N and R can be formulated as follows: "a delimiter tag "0" will be preceded and followed by a positive integer, while a "0" representing a difference of 64 will be either preceded or followed by another "0". For example the sequence "006" will represent a difference of 70 (((2-1)*64)+6), while the sequence "706" will contain a delimiter tag. The number of the first document where the descriptor occurs is stored in START1.

The relation between DICT1, INV1, START1 and ININV1 can be schematized as shown in Fig. 6.

Organization of Level2.

In the second level of the storage hierarchy DICT2 contains the next most frequent 1,024 descriptors not already stored in DICT1. There is also an index table INDICT2 and a corresponding inverted file INV2. All the descriptors stored in this level are assigned a 12-bit code "1xxxxx0xxxxx" where each x denotes a 0 or 1. The first 6 bits form an address to a pointer table PT2 where the length of the descriptor is stored, and the last 6-bits give the position in DICT2. The number of the first document in which a descriptor occurs is stored in START2.

Two hashing functions H1 and H2 which have hash values between 0 and 31 may be used to address an element INDICT2 (H1(c1),H2(c2)) and ININV2(H1(c1),H2(c2)). Each element of ININV2(H1(c1), H2(c2)) contains a pointer of 20 bits which points to the list of the documents where it occurs, this list is stored in INV2. The number of the first document of the list will be stored in START2 in a 20 bit location.

The first element stored in the inverted lists (INV2) will contain a backward pointer on ININV2 which will serve during the update process. It is necessary because when a document is added or deleted the length of each inverted file associated with a descriptor in this document will shift and this will change the starting addresses of all the trailing inverted files. In ININV2 is the pointer that indicates the head of the list. This address will be the 12 bit code stored in the compressed data base. Every element of INDICT2 is a 10 bit record, 5 bits for the position of a descriptor in a row of DICT2, and 5 other bits which indicate an address in PT2 where the length of the descriptor is stored. At this location is also stored a pointer to a row of DICT2 where other descriptors of the same length are stored, a flag bit will indicate whether that row is full.

Using the formula (1) it may be estimated that the proportion of terms in the data base text that appear in DICT2 is equal to

$$P2=(LN(1056) - LN(32))/(LN(D)+G)=3.48/(LN(D)+G$$

thus the relation between D and P2 is as follows:

D	P2
12,000	0.35
50,000	0.31
200,000	0.27

Applying the formula (2) to this level it is to be expected that in the inverted file INV2 the average difference

between successive document numbers is

$$DELTAV2 = (N/T)*(LN(D)+G)/(1/I)$$

$$=((1024(LN(D))+G)/(LN(1056)-LN(32))*(N/T)$$

$$=(1024)/P2T$$

Hence for the last list in INV2 the average difference between successive document number is

$$DELTA(1056)=1056(LN(D)+G)*(N/T)$$

The relation between D, T, DELTAV2, DELTA1056 and the length L1056 of the last list is summarized in the table below

D	T=10N	DELTAV2	DELTA(1056)	L1056
12,000	100,000	293	1,000	12
50,000	1,000,000	320	1,100	45
200,000	10,000,000	380	1,200	175

If the document numbers in INV2 are each stored in 20 bits the total storage required for INV2 is 20*P2*T bits. Alternatively if difference coding of 12 bits is used for each document number the space required for storage of INV2 is

12*P2*T BITS = 2*P2*T characters.

The interconnection between the different parts of the level is represented in Fig. 7. FREE2 is a bit array which indicates

whether the corresponding position in DICT2 is empty. It will be mainly used when a term is removed from the data base.

Storage Required for Level2.

INDICT2: 1024*10 = 10240 bits.

PT2: 32*11 = 352 bits.

ININV2: 1024*20 bits = 3414 chars. (CDC)

START3: 1024*20 bits = 3414 chars. (CDC)

INV2: 2P2T chars.

DICT2: 1024*average length of the 1024 descriptors.

Organization of Level3.

The third level of storage contains the next most frequent 32,768 descriptors in DICT3(K) which has an index-table INDICT3(K) and a corresponding inverted file INV3(K). This level is in fact subdivised into 32 sublevels. All the descriptors stored in this level have an 18-bit code of the form "1XXXXX1XXXXX0XXXXX" where the first 6 bits represent the sublevel number, and the remaining 12 bits have the same meaning as the code in the level2. To each sublevel is associated an array element START3(K) in which is stored the number of the first document where a descriptor occurs. The value of K is determined by a hashing function H3(C2) = H2(C2). Each component of a sublevel (K) has the same role, and contains the same information, as the corresponding part in level2.

Using the formula (1) it may be predicted that the proportion of data base text that appears in DICT3(K) K=0..31 is

$$P3=(LN(33824)-LN(1056))/(LN(D)+G)=3.45/(LN(D)+G)$$

thus based on $\sum_i P_i=1$ the relation between P3 and D is:

D	P3
12.000	0.25
50.000	0.31
200.000	0.27

The proportion of data base text represented by DICT1, DICT2, DICT3(K) is:

D	Pi
12.000	1.0
50.000	0.98
200.000	0.86

But the proportion of text in DICT3(K) can be increased by changing the range of the hash value. An increase in the value of K changes the number of sublevels and hence raises the upper bound of the system. But one has to be aware that this will increase the code assigned to the less frequent descriptors, and hence decrease the efficiency of the compression scheme.

Applying the formula (2) to this level, it may be expected that in the inverted file INV3(K) the average difference between successive document numbers is:

$$DELTAV3=((32768(LN(D)+G)/(LN(33824)-LN(1056))*(N/T)$$

$$DELTAV2=32768N/P3*T$$

so for the last list in INV3 the average difference between successive document numbers is DELTA33824 where

$$DELTA33824=33824(LN(D)+G)*(N/T)$$

The relation between D, T, DELTAV2, DELTA33824 and L (the length of the last list) is summarized in the following

table:

D	T=10N	DELTAV3	DELTA33824	L33824
12,000	100,000	-	-	-
50,000	1,000,000	10,500	35,200	1.4
200,000	10,000,000	12,500	38,400	5.3

If the document numbers in INV3 are each stored in 20 bits the total storage required for INV3 is 20 P3*T bits. Using difference coding of 16 bits the required storage for INV3 is:

$$16*P3*T \text{ bits} = 2.7*P3*T \text{ characters.}$$

$$= 0.84*T$$

which if allocated between 32 dictionaries requires each dictionary DICT3(K) to store .085 P3T characters.

T	.085P3T
1,000,000	26,000
10,000,000	230,000

Storage Required for Level3.

The storage required for LEVEL3 is equal to

32* storage required for level2.

A problem arises concerning the space to be alloted for the dictionary. Because the CDC computer is a word-oriented computer it is not convenient to allocate space in characters without reference to word length, and so for purposes of dictionary storage the descriptors may be grouped into three classes:

1) length less than 10 characters.

2) length between 11 and 20 characters.

3) length between 21 and 30 characters.

The average wasted space has been estimated with five different data bases (Table 11) and the results are presented in the Tables 5 to 7.

In order to generate dynamically the exact space needed for each line of DICT the space required for the array is fetched in a 'space stack', and when the table will be constructed, it will be stored in a file, and the space alloted for its construction will be released and replaced in the 'space stack'.

Generation of the dictionary is achieved by the inverse process:

a) pop up the exact space required to store the dictionary.

b) bring the dictionary file into this space, so the only wasted space is for the blanks needed to pad words that do not fill a computer word. The only wasted space will then be for the blanks needed to pad words that do not fill a computer word.

Using the Table 11 which gives the proportion of title

terms of length L in different data bases, a tabulation may be made as follows of the overhead due to the word-addressing mode. It is understood that this overhead would not exist if an assembler language were used but the programming work would be more tedious. The storage cost is tabulated below.

	D	Storage(char)	overhead	%
kf	50,406	3,279,516	139,146	4
ct	63,316	4,518,863	206,261	5
mt	31,004	1,874,688	76,886	4
ca	15,907	1,046,204	43,961	4
cd	10,804	730,891	32,230	4

6.6 Compression Rate

In the Kucera Francis table the 32 most frequent words have an average length of 2.68 chars. In the compressed data base each corresponding descriptor is stored in 1 byte, and hence the compression ratio in LEVEL1 is (1/2.68) = 0.37. Within the LEVEL2 if the 1,024 descriptors have an average length of 6.6 chars, and are each coded in two bytes in the compressed data base the resulting compression ratio in LEVEL2 is (2/6.6) = 0.30 and in LEVEL3 is (3/6.6) = 0.45. Then the overall compression ratio for descriptors in DICT1, DICT2, DICT3 is:

$$(P1T(0.37)+P2T(0.30)+P3T(0.46))/(P1T+P2T+P3T)$$

$$=0.38$$

6.7 Compressed Data Base

The compressed data base contains the computer files of bibliographic information stored in a compressed form. The compressed data base will be a succession of coded terms, each of which has the form "0xxxxx", "1xxxxx0xxxxx", or "1xxxxx1xxxxx0xxxxx". It is supposed that each record of the compressed data base contains:

1) a title field.

2) an abstract field.

3) an author field that contains:

   i) name.

   ii) first name

   iii) address of the author.

4) a document field that contains:

   i) a "1-bit" code indicating the category of the document:

      "0" = book.

      "1" = article within a journal or proceeding.

   ii) a coden field (LC number for a book or coden for a journal).

   iii) a page number. If the document is a book then this number indicates its total number

of pages. If it is a document then the first (n-2) digits indicates the page where the document starts and the last two digits indicate the total number of pages. For example (34505 denotes a document of length 5 pages, whose starting page is 345). Two digits have been chosen because it is rare that an article appearing in a periodic or proceeding has more than 99 pages. If this situation occurs we store the starting page followed by "00" and then the page length. For example if an article has 120 pages and starts at page 345, his code will be "34500V 120".

IV) a language field, which is a 3-bit code indicating the language in which the article is written.

"000" = english

"001" = french

"010" = german

"011" = russian

"100" = spanish

"101" = italian

"110" = japanese

"111" = other

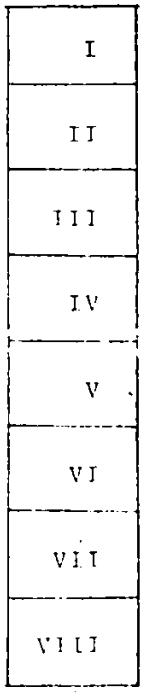
POOR COPY

V) a date field, which is the date of publication.

VI) a published field. (initials and address of the publisher)

VII) the "internal" or "local" library code. If the article or book does not exist in the library this field is flaged with "nil".

VIII) a keywords field.



Data Structure associated with a bibliographic record.

example

a) a book record could be represented as follows:

"operating system principles"

"this book tries to give students of computer science and professional programmers a general understanding of operating systems. The programs that enable people to share computers efficiently"

POOR COPY

Hansen P.B. California Institute of Technology
0 QA76.5B76 366 00 73 P.H. New Jersey nil

1) time sharing computer system. 2) computer programming management

b) an "article record" could have this representation.

"a document storage method based on polarized distance"

"some elementary mathematical properties of term matching document retrieval system are developed. These properties are used as a basis for a new file organization technique. Some of the advantages of this new method are (1) the key-to-address transformation is easily determined; (2) the documentary information is stored only once in the file; (3) the file organization allows the use of various matching functions and thresholds; and (4) the dimensionality of the transform is easily expanded to accomodate various size data bases"

Chien R. T. M.I.T, Cambridge, Mass.
Mark E.A. University of Illinois at Urbana-Champaign, Urbana, Ill.
1 jacm 23312 00 74 acm 5th Ave. N.Y.

Document retrieval, file organization, term matching retrieval, search method, binary indexing, matching function.

6.8 Strategy for Retrieving a Term in the Dictionaries

Two parameters are considered when we want to retrieve a descriptor 1) its length 2) its content.

POOR COPY

When a descriptor is read and is longer than the longest element in DICT1, this descriptor is not compared to any element of this dictionary. But by use of some fragments of it (C1 and C2) an address (I) in INDICT2 will be computed. At this address is stored the location in PT2 that contains both length of all the descriptors in that row. If the lengths are the same, the descriptor is compared to the content of a particular location in INDICT2, otherwise we look for another address in INDICT2, according to the collision process.

When all the elements of this level have been examined we restart at a lower level but at the same position that we began the examination at the upper level. The search is terminated if 1) we find the descriptor, or 2) an empty location in INDICT2 is reached, or 3) the last position at the lowest level of INDICT2 has been examined.

During the decoding process the strategy is less tedious because the code contains 1) the address in PT2 where is stored the pointer of the particular row of DICT2 2) the column of that row where the descriptor is stored and the level in the hierarchy if it is necessary.

The use of a hierarchical storage gives some advantages. The location of the files on the support is partitioned in 32 zones Z1, .. .,Z32. The hash function (H2oH1) gives for each descriptor an address in Z2. If a collision occurs at this location, a translation is executed which allows us to examine a position in Z3. The process is repeated until an empty location is reached.

As all the different zones Zi have the same size 1024, the general formulation for finding a location is:

$$FI = H2(C2) \circ H1(C1) + 1024*i \qquad I=1,\ldots,31$$

But at each iteration the probability of collision decreases. Taking that into account we could allot decreasing sizes for the different zones, so a free location could be searched according to the formula

FI = H2(c2)oH1(c1) + 1024*a(i)*i with a(i+1)<a(i) .. a(0)<1

The sizes of the levels in the hierarchy could constitute a decreasing geometrical progression, but by using fixed size we reduce the probability of collisions.

CHAPTER 7

DESCRIPTION OF OVERALL RETRIEVAL SYSTEM

The system whose flowchart is given in appendix D can be decomposed into two sub-systems.

a) the automatic file compression system used only in creation of the files.

b) the query-answering system.

The automatic file compression system contains four separate components: an analyser, a compressor, a constructor and an expander. While the query-answering system is composed mainly of an expander. The analyser automatically scans the file and computes the statistics on the frequency of the words in the data base, the compressor codes the data according to the proposed compression scheme, while the constructor builds the compressed data base and the inverted files that will be used by the expander to reconstruct the original text by recovering the adequate descriptor after a query has been processed.

7.1 The Analyser

The analyser will (1) scan the entire file of bibliographic elements (2) find the frequency of each word and produce a list of all the different words of the data base in descending order of their frequency in that particular data base. It may be noted that:

1) for any particular data base the analyser need be used only at infrequent intervals depending on the rate at which the statistical properties of the data base change.

2) the data base need not be reanalysed for every (or small amount of) change, although extensive changes might well make reanalysis desirable in order to avoid loss of coding efficiency.

3) the output produced by the analyser can be used to identify and validate the information stored in the data base.

The main disadvantage of using such an analyser is that it cannot be used for part of the data. In order to optimize the code generated by the compressor it is necessary to scan the entire data base.

7.2 The Compressor

The compressor receives as input the file generated by the analyser and it creates the large dictionaries that contain each descriptor that appears in the data base. All the pointer tables are generated. For each descriptor the compression code will be a function both the position of the descriptor in the dictionary and the level of that dictionary in the hierarchical storage. The advantages of this scheme are:

1) the compressor code does not depend in any way on the machine used and in a sense on any prior knowledge of the files to be compressed.

2) the code generated is a variable length code and it is almost optimum because there is no redundancy. The only way that optimality is lost is when a frequent descriptor is stored at a lower level because there is not a free position of that length within the empty buckets that still remain in the dictionaries of an upper level. But this disadvantage is minimized by

the fact that this descriptor will be accessed in a single access when it is looked for in the level where it has been placed.

3) the documentary information is stored in uncompressed alphanumeric form only once.

4) the code generated for each descriptor is unique, and there is no danger of overhead produced by a bad choice of stem or cord.

The compressor is the basic module of the system that will be used whenever new data is inserted in the compressed data base. It is understood that each new term will be placed in the first available place that could receive it. Thus it will not be placed automatically after the last element previously entered.

7.3 The Constructor

The main function of this module is to build the compressed data base, the inverted files, and the pointer tables associated with them. In order to generate the code placed in the inverted files to specify in which field of the document the descriptor occurs the different fields of each document are specified by a special symbol. Each document will constitute a segment of the compressed data base.

As far as the compression part is concerned, this module achieves essentially the same task as the compressor but its main task is to build the inverted files required during the retrieval process. During the compression phase only the dictionary file and the pointer table file are used.

7.4 The Expander

This procedure is presented with the compressed data base (C.D.B) as input and, using the compression code, recovers original uncompressed alphanumeric information. The procedure can be subdivised into the following two sub-procedures.

The "query-decoder", that analyses and evaluates the query. For this operation only the inverted files are required.

The "translator" that reconstructs the original information. It uses only the dictionary file and the pointer tables associated with them.

The inverted file search is a two-step process; First, the matching document's accession numbers are retrieved from the inverted file, and then the documentary information is retrieved from the primary file (C.D.B.) using the accession numbers as addresses.

The expander module can be run either in interactive, for a simple query, or in batch mode for a more elaborate query. The results of the query evaluation will be written on a scratch file to allow "browsing", in other words people can examine the results of their query on line, so they can change the form of their query is the results are not satisfactory.

7.5 Update

The principal reason for on-line real-time update of a file is to serve the operational needs of those systems in which the update transaction must be usable in the file within a very short time from its declaration. Evaluation of updating cost is very hard to make because there is no real clear-cut separation between the hardware and the software cost.

POOR COPY

Lefkovitz (43) gave a formula to evaluate update cost either in on-line or batch mode. In updating an information system one has to distinguish five main categories of operation:

1) whole record addition which consists of:

   a) edit record for file entry.

   b) assign an address on the direct access device to the new record.

   c) decode key I in the directory and the inverted list.

   d) insert ad, in sequence, into the list. Solve overflow if it occurs.

   e) increment list length and fixe pointer in the inverted list.

   f) repeat c through e for all keys in the new record.

   g) store new record at the designated address.

2) whole record deletion, which consists of:

   a) set record deletion bit.

   b) decode every key of the record in the directory, remove record address, from respective inverted lists and decrement list length by 1.

3) deletion of keys.

4) addition of keys.

POOR COPY

If an inverted file organization is adopted the last two processes are used only when it is desired to change the inversion percentage of the file, as for example in passing from a fully inverted file to a partially inverted one and vice versa.

Two main problems have to be considered in relation to the update process. The first one is update of the directory and change of the list pointer whenever a key is involved. The second concerns the relocation of a record when an update expands its size, and the subsequent utilization of its former spaces. These two processes are very cumbersome if an inverted file organization has been choosen. But if the record accession numbers, rather than the addresses, are stored in the inverted lists then when a file record is relocated none of the key list has to be modified; only a single modification to the accession number decoder is necessary. The retrieval process, however, is considerably less efficient. The list of relevant documents obtained after processing the query will consist of a sequence of accession numbers each of which must then be decoded to an address. This list must be kept on a file to be scanned during the output process.

During the updating process two main operations have to be considered 1) addition of new elements 2) deletion of useless elements, both operations require the update of the following files:

1) dictionaries.

2) inverted files.

3) compressed data base.

4) file "START".

POOR COPY

7.5.1 Addition

When a new document is to be added any new descriptors that belong to this document, but do not already appear in the dictionaries, must also be added. This is achieved by scanning each storage of the hierarchy until there is found a free bucket of appropriate length. If such a bucket does not exist it must be created in the first available dictionary. The corresponding bit of the bit-array "free" is set to "1".

The updating of the inverted file is more tedious, when adding a new element two different types of operation have to be made on the inverted files.

1) update of the inverted list or descriptors that are already present in other documents.

2) creation of new lists for new descriptors not previously present.

During the process of updating the inverted files both the inverted files (INV1) and the corresponding pointer tables (ININV1) may need to be updated.

The first step of the process is to update the already existing inverted lists. Each time a new element is added to such a list the pointer table ININV1 is scanned and the values of the pointers that point on the subsequent lists are increased by one. This process is repeated for all the inverted lists concerned with the update. During the second step the new descriptors that appear the vocabulary of the data base are placed at the "bottom" of the inverted file and the appropriate pointers are set, as are the corresponding positions in "START".

POOR COPY

7.5.2 Deletion

The deletion process is done by the inverse process. First the bit array free is set to zero. That has for effect to clear the associated positions in DICT, INDICT, ININV and START. The file INV is updated by recopying the old file into a new one in which the lists associated to the unused keys have been deleted. The value of the pointers that were pointed below the deleted ones are decreased by a number equal to the length of the deleted list. These operations are repeated for each deleted keys. The appropriate record is deleted in the compressed data base. The content of the record is replaced by the message "deleted" in order to maintain the same sequence of accession number. Because if the sequence is changed all the inverted lists have to be updated. This operation would be very expensive.

7.6 A Proposed Query Language

A efficient query language for the system should have two essential characteristics. First it must be easy to use and allow searches to be performed on author or subject. Three different classes of commands can be used: a) author commands b) subject commands c) printing commands.

7.6.1 List of Author Commands

"A=" will list all the documents written by an author.

The syntax of the command is ⟨A=⟩ ::= ⟨Name⟩ | ⟨Name, First name⟩| ⟨Name,Firstname,Initial⟩|⟨Name,Initial,Initial⟩

"P=" will list all the documents about an author.

The syntax of the command is: ⟨P=⟩ ::= ⟨Author⟩

Example: P= Nixon, R will list the documents about R Nixon but not the documents written by R Nixon.

"X=" will list the documents written by an author and those related to him.

The syntax of the command is ⟨X= ⟩ ::= ⟨Author ⟩

Example: X= Nixon, R will produce the same results as A= Nixon,R and P= Nixon,R.

"C=" will list the document written by a society.

The syntax of the command is ⟨C= ⟩ ::= ⟨Sigle⟩

Example: C= IBM will produce the documents written by the International Business Machine.

7.6.2 List of Subject Commands

One can list all the documents related to a subject term by simply writting the descriptor associated with the term.

Example: "COMPUTER" will list all the documents related to computer.

Questions may be formulated by use of the logical operators "AND", "OR", "NOT". The priority of the operator is the usual left to right. This sequence can be broken by use of parentheses.

7.6.3 Lists of Impression Commands

"=" will print on the terminal a fixed number (n) of documents related to the query. A default value of 5 can be assigned to this number. This fixed number of document will be printed.

out each time a "CR" is performed. The user has the opportunity to change this number by use of the command "$ln", where n is the number of document desired.

"$p" will display the entire answer-file on a line printer. A maximum number of lines is allowed and a message will be printed if this number is exceeded.

7.6.4 Truncation

Two categories of truncation are allowed. First the right or left truncations "*" which are very usefull when the correct spelling of a word is unknown. For example the query X= Dosto* will generate the documents related to Dostoievsky. Similarly compute* will generate documents related to compute or computer, or computation, etc.

The other form of truncation is the truncation with dictionary "&". A descriptor is selected and the user can accept or reject this descriptor by typing "y" or "n". For example if the keyword is "Econo&", the descriptor "Econometric" can be accepted by typing "y". A second descriptor "Econometrica" can be suggested, by typing "n", the descriptor will be rejected. On issue of a "cr" not preceded by "n" or "y" then automatically accept the last descriptor and all the others that remain in the list. The command "t" will print the entire list of synonyms for a descriptor.

Both truncations can be applied either to author names or subject terms.

7.7 Conclusion

In conclusion, a compact file organization has been presented. A high degree of compression has been achieved in

POOR COPY]

certain cases. Because of use of hashing as a randomizing tool for the retrieval process, the retrieval time is relatively fast. The double hashing technique used, without decreasing the efficiency of the hashing scheme, allows front and/or rear truncation which means that fragments can be used as query elements. Because of the modularity of the file organization we can associate with each module a dedicated microprocessor which will perform the search process in each module. The microprocessor to be activated will be determined by the hashing function which selects the module (K). It could happen because of the collision that the descriptor is in a lower module, to speed up the retrieval process we could use simultaneously all the microprocessors from k to 32. The CPU cost of the file organization has not been evaluated. It will vary depending on the data base used and on the parameters that have been described previously. Thus the file organization could be simulated in order to evaluate the performance in any particular case.

POOR COPY

REFERENCES

1. Tillit, Halley E.: "An Experiment in Information Searching with the 701 Calculator".
Journal of Library Automation, 3:(202-206), September 1970.

2. Bracken, R.H.: Tillit, H.E.: "Information Searching with the 701 Calculator".
J.A.C.M., 4:(131-136), April 1957.

3. Tillit, H.E.: "An Application for an Electronic Computer to Information Retrieval".
Modern trands in Documentation, (67-68), N.Y. 1959
Pergamon Press.

4. Bracken, R, Oldfield, Bruce: "A General System for Handling Alphanumeric Information in IBM 701 Computer".
J.A.C.M., 3:(175-180), July 1959.

5. Barton, A.R., Schwatz, V.: "Information Retrieval on a High Speed Computer".
Evendal, Ohio: General Electric Co. pg 8, 1959.

6. Austin, C.: "Medlars".
Bethesda, Maryland: National Library of Medecine, pg 1963-67, 1968.

7. Summit, Roger: "Dialog, and Operationnal On-Line Reference Retrieval System".
In A.C.M. Proceedings of 22nd National Conference in Washington, D.C., pg 51-56.

8. Pizer, I.: "Regional Medical Library Network".
Bulletin of the Medical Library Association, pg. 101-115, April 1969.

POOR COPY

9. Bunnow, L.R.: "Study of an Proposal for a Mechanized Information Retrieval System for the Missiles and Space Systems Engineering Library".
Santa Monica California, 1960.

10. Kilgour, F.: "History of Library Computerization."
Journal of Library Automation, 3:(218-229), 1970.

11. Warheit : "File Organization of Library Records".
Journal of Library Automation, 2:(20-31), 1969.

12. Mitchell, P., Burgess, I.: "Methods of Randomization of
13. Large Files with High Volatility".
Journal of Library Automation, 3:(79-87), 1970.

14. Dimsdale, J.J., Heaps, H.S. : "File Structure for On-Line Catalog of One Million Titles".
Journal of Lib. Aut., 6:(37-55), March 1973.

15. Warheit : "File Organization of Library Records".

16. Salton, G., Lesk, E. : "The Smart Automatic Document Retrieval System - an illustration.
C.A.C.M., 8:(391-398), June 1969.

17. Chien, R.T., Mark, E. : "A Document Storage Method Based on Polarized Distance".
J.A.C.M., 21:(233-245), April 1974.

18. Schay, G., Raver, N. : "A Method for Key-to-Address Transformation".
IBM J. Res. Develop., 7:(121-126), April 1963.

19. Buchholtz, W. : "Film Organization and Addressing".
IBM Systems Journal, 2:(86-111), June 1963.

POOR COPY

20. Maurer, W.D. : "An Improved Hash Code for Scatter Storage".
C.A.C.M., 11:(35-36), January 1968.

21. Mealy, G.H. : "A Generalized Assembly System".
McGraw Hill, pg 550, N.Y. 1957.

22. Morris, R. : "Scatter Storage Techniques".
C.A.C.M., 11:(38-43), January 1968.

23. Bell, J. : "The Quadratic Quotient Method - A Hash Code Eliminating Secondary Clustering".
C.A.C.M., 13:(107-109), February 1970.

24. Radke, C. : "The Use of Quadratic Residue Research".
C.A.C.M., 13:(103-105), February 1970.

25. Hopgood and Davenport: "The Quadratic Hash Method when the Table Size is a Power of Two".
Computer Journal, 15:(314-315); 1972.

26. Day, A.C. : "Full Table Quadratic Searching for Scatter Storage".
C.A.C.M., 8:(481-482), August 1970.

27. Bell, J., Kaman, C.H. : "The Linear Quotient Hash Code".
C.A.C.M., 13:(675-677), November 1970.

28. Luccio, F. : "Weighted Increment Linear Search for Scatter Storage".
C.A.C.M., 15:(1045-1047), December 1972.

29. Lum, V.Y., Yuen, T., Dodd, M. : "Key-to-Address Transfer Techniques - A Fundamental Performance Study on Large Existing Formatted Files".
C.A.C.M., 14:(228-239), April 1971.

POOR COPY

30. Rubin, F.: "Experiments in Text File Compression."
C.A.C.M., 19:(617-623), November 1976.

31. Ruth, S., Kreutzer, P. : "Data Compression for Large Business Files".
Datamation 18:(62-66), September 1972.

32. Kusmiss, J.M. : "An Experiment in Adaptive Encoding".
IBM Tech Report, N.Y. 1974.

33. Hagamen, W.D., et al. "Encoding Verbal Information as Unique Number".
IBM System Journal 11:(278-315), October 1972.

34. Gottlieb, D., Hagerth, S., Lehot, P. : "A Classification of Compression Methods and Their Usefulness for a Large Data Processing Center".
National Computer Conference, 1975, 453-458.

35. Svank, M.I. : "Optimizing the Storage of Alphanumeric Data".
Canadian Datasystem, May 1975, pg 38-40.

36. McCarthy, J.P. : "Automatic File Compression".
International Computing Symposium 1973, pg 511-516.

37. Schieber, T. : "An Algorithm for Data Compression".
Journal of Library Automation, August 1971, pg 204.

38. Heaps, H.S. : "Storage Analysis of a Compression Coding for Document Data Bases".
Infor., 10:(47-61), February 1972.

39. Reid, W., Heaps, H.S. : "Compression of Data for Library Automation".
Automation in Libraries, 1971, pg 2.1-2.21.

POOR COPY]

40. Schuegraf, E., Heaps, H.S. : "A Comparaison of Algorithms for Data Base Compression by Use of Fragments as Language Elements".
Information Storage and Retrieval 10;(309-319), 1974.

41. Cardenas, A. : "Evaluation, Selection of File Organization. A Model and System".
C.A.C.M., 16:(540-548), September 1973.

42. IBM System/360, P1/1 Ref. Man.

43. Lefkovitz, D. : "File Structure for On-Line Systems".
Spartan Books, 1969.

44. Yau, S.S., Fung, H.S. : "Associative Processor Architecture - A Survey".
A.C.M. Computing Surveys 9:(3-27), March 1977.

TABLE 1

AVERAGE NUMBER OF ACCESSES

Sample Number	a = 100% (32/32)	a = 75% (24/32)	a = 72% (23/32)	a = 75% (32/43)
1	3.18	2.00	2.04	2.72
2	3.37	1.96	1.78	2.12
3	3.18	2.23	1.91	2.25
4	3.34	2.00	1.83	1.75
5	3.46	2.00	1.95	1.90
6	3.18	2.63	2.60	2.00
7	3.68	1.71	1.73	1.71
8	3.87	3.00	2.73	2.21
9	2.62	2.04	2.08	1.68
10	3.78	2.08	2.00	2.37
11	4.71	2.38	2.21	2.03
12	2.84	2.00	1.95	2.18
13	3.40	1.75	1.78	2.03
Mean	3.43	2.14	2.03	2.07

The results obtained by Lum (9)

Method Division

Bucket Size=1

Overflow: Open addressing

(Linear Search)

a=70%	4.73
a=75%	7.20
a=95%	25.73

TABLE 2

TOTAL NUMBER OF COLLISION

	a = 100% (32/32)	a = 75% (24/32)	a = 72% (23/32)	a = 75% (32/43)
1	70	24	24	55
2	76	23	18	36
3	70	31	21	40
4	75	24	13	24
5	73	24	22	23
6	70	33	37	32
7	86	17	17	23
8	92	48	40	33
9	52	25	25	22
10	83	26	23	44
11	119	33	28	33
12	59	24	22	38
13	77	18	18	33
Mean	78	27.38	24.55	34.46

Results obtained

by Lum

Bucket size 1

Method Division

a=70% 25 col

a=75% 25 col

a=95% 33 col

TABLE 3

LONGEST SEQUENCE OF COLLISION

	a = 100% (32/32)	a = 75% (24/32)	a = 72% (23/32)	a = 75% (32/43)
1	9	5	5	10
2	16	5	4	9
3	10	10	6	8
4	16	5	4	5
5	31	4	4	4
6	7	6	6	5
7	31	8	8	9
8	13	9	9	5
9	11	4	4	4
10	28	7	7	6
11	30	8	8	6
12	14	5	5	8
13	16	4	4	9
Mean	17.08	6.0	5.7	6.7

TABLE -

TOTAL TIME (EXECUTION)

	a 100% (32/32)	a 75% (24/32)	a 72% (23/32)	a 75% (32/43)
1	0.21	0.19	0.21	.11
2	0.23	0.18	0.22	.11
3	0.18	0.19	0.24	.10
4	0.14	0.19	0.19	.10
5	0.24	0.20	0.19	.10
6	0.21	0.19	0.19	.10
7	0.21	0.19	0.23	.12
8	0.21	0.20	0.24	.10
9	0.19	0.19	0.21	.10
10	0.20	0.18	0.21	.10
11	0.21	0.18	0.21	.10
12	0.19	0.19	0.21	.11
13	0.22	0.18	0.20	.10
Mean	0.20	0.18	0.20	.10

TABLE 5

AVERAGE STORAGE FOR DICT2 FOR A DATA BASE OF 50,000 DIFFERENT WORDS

Length	E (Total number of words of length L)	E (Number of these words in each level) — Col. 2 / 34	Average E (Number of row of that length in each level) Col. 3 / 32
1	40	1.21	.04
2	450	15.15	.47
3	1350	65.45	1.42
4	3150	92.64	2.89
5	4370	128.52	4.01
6	5650	166.17	5.19
7	6380	187.64	5.86
8	6630	195.97	6.09
9	6210	182.64	5.70
10	5170	152.05	4.75
11	3700	108.82	3.40
12	2640	77.64	2.42
13	1750	51.47	1.60
14	1050	30.88	.96
15	310	9.11	.28
16	160	4.85	.15

TABLE 6

MINIMUM STORAGE FOR DICT2

1	0	0	0
2	250	7.58	0.24
3	1050	31.82	0.99
4	2200	66.67	2.08
5	3100	93.94	2.34
6	4700	142.42	4.45
7	5250	153.09	4.97
8	6300	190.91	5.37
9	5750	174.24	5.45
10	4450	134.85	4.21
11	2950	89.39	2.79
12	1850	56.06	1.75
13	1150	34.85	1.09
14	600	18.18	0.57
15	350	10.61	0.33
16	150	4.55	0.14

POOR COPY]

TABLE 7

MAXIMUM STORAGE FOR DICT2

1	100	3.03	0.03
2	600	18.18	0.57
3	1650	50.0	1.56
4	4200	127.27	3.38
5	5400	163.64	5.11
6	6750	204.55	6.39
7	7100	215.15	6.72
8	6300	205.03	6.53
9	6800	206.06	6.44
10	5350	180.30	5.63
11	4000	121.21	3.79
12	3300	100.00	3.13
13	2400	72.73	2.27
14	1500	45.43	1.42
15	650	19.70	0.62
16	350	10.61	0.33

TABLE 8

RESULTS OBTAINED BY LUM FOR THE DIVISION HASHING METHOD

Load Factor	50%	55%	60%	65%	70%	75%	80%	85%	90%	95%
Mean Number of Access	4.52	5.37	5.04	8.84	4.73	7.20	10.10	13.27	22.42	25.79
Number of Collisions	17	21	20	27	25	25	28	30	29	33

TABLE 9

RESULTS OF THE HASHING SCHEME USING THE FRAGMENTS C1

														Means	B.S
No. of Collisions	75	68	92	113	83	93	18	80	29	49	24	35	105	66.46	
Longest Sequence of Search	15	13	23	24	20	20	6	14	11	8	5	11	23	14.84	10
Mean Number of Search	3.34	3.12	3.87	4.53	3.59	3.90	1.9	3.5	2.26	3.04	2.30	2.45	4.28	3.23	5

Longest Search (c1, c2) = 20

Mean Search (c1, c2) = 4.42 $\angle$ 5.0

TABLE 10

RESULTS OF THE HASHING SCHEMES USING THE FRAGMENTS C2

No. of Collisions	165	173	64	124	168	256	30	220	108	70	24	110	303	Means 139.61	B.S
Longest Sequence of Search	30	22	10	31	29	27	6	28	15	15	6	14	31	20.30	10
Mean Number of Search	6.15	6.40	3.0	4.87	6.25	9.0	2.5	7.85	5.69	3.91	2.04	5.58	10.46	5.66	5.0

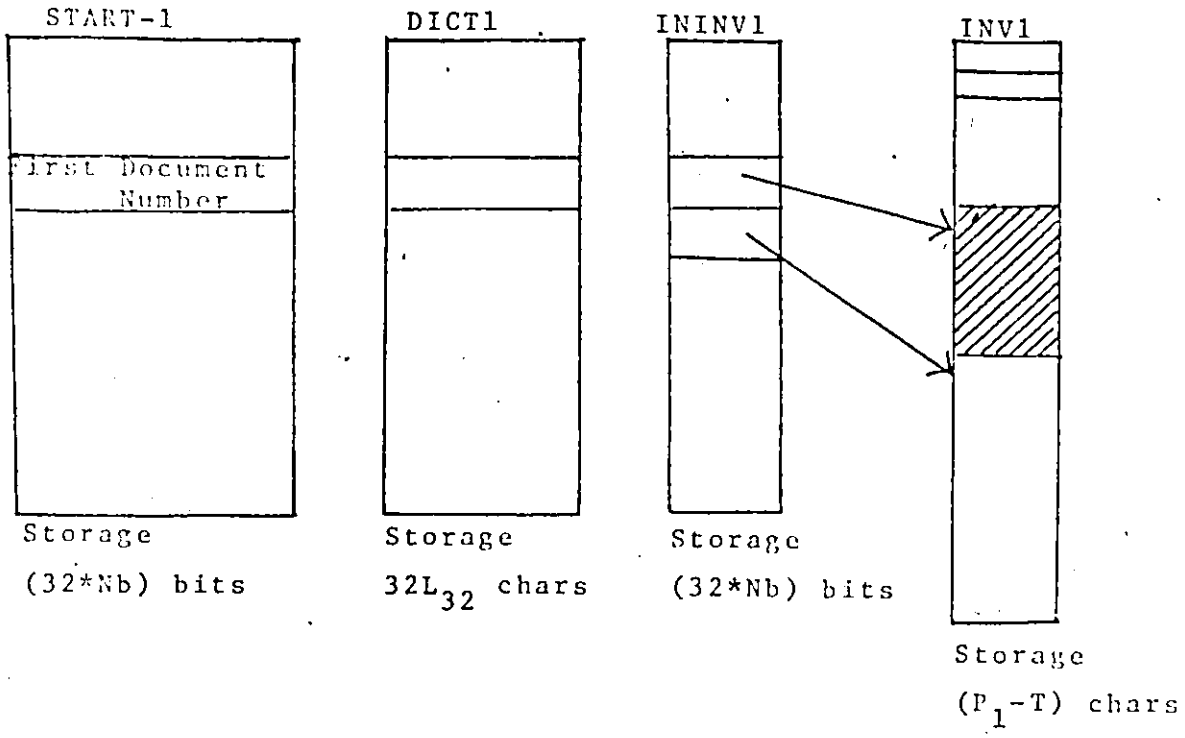
Longest Search (c1, c2) = 20

Mean Search (c1, c2) = 4.42 < 5.0

TABLE 11

PROPORTION OF WORDS OF LENGTH L IN DIFFERENT DATA BASE

L	K and F D=50,406	CT63-65 D=63,316	MARC01-58 D=31,004	CAIN D=15,907	CANDICT D=10,804
1	.001	0.	.001	.002	0.
2	.005	.011	.011	.012	.006
3	.024	.029	.033	.028	.021
4	.060	.044	.084	.067	.060
5	.092	.062	.108	.091	.084
6	.128	.094	.135	.103	.105
7	.145	.105	.142	.128	.118
8	.138	.126	.137	.129	.133
9	.120	.136	.115	.123	.127
10	.096	.119	.089	.102	.111
11	.068	.091	.059	.072	.080
12	.046	.066	.037	.056	.059
13	.029	.048	.023	.037	.038
14	.020	.030	.012	.020	.023
15	.011		.007	.013	
16	.007		.003	.006	



Nb Number of bits required to represent the maximum document number.

Fig. 6

POOR COPY]

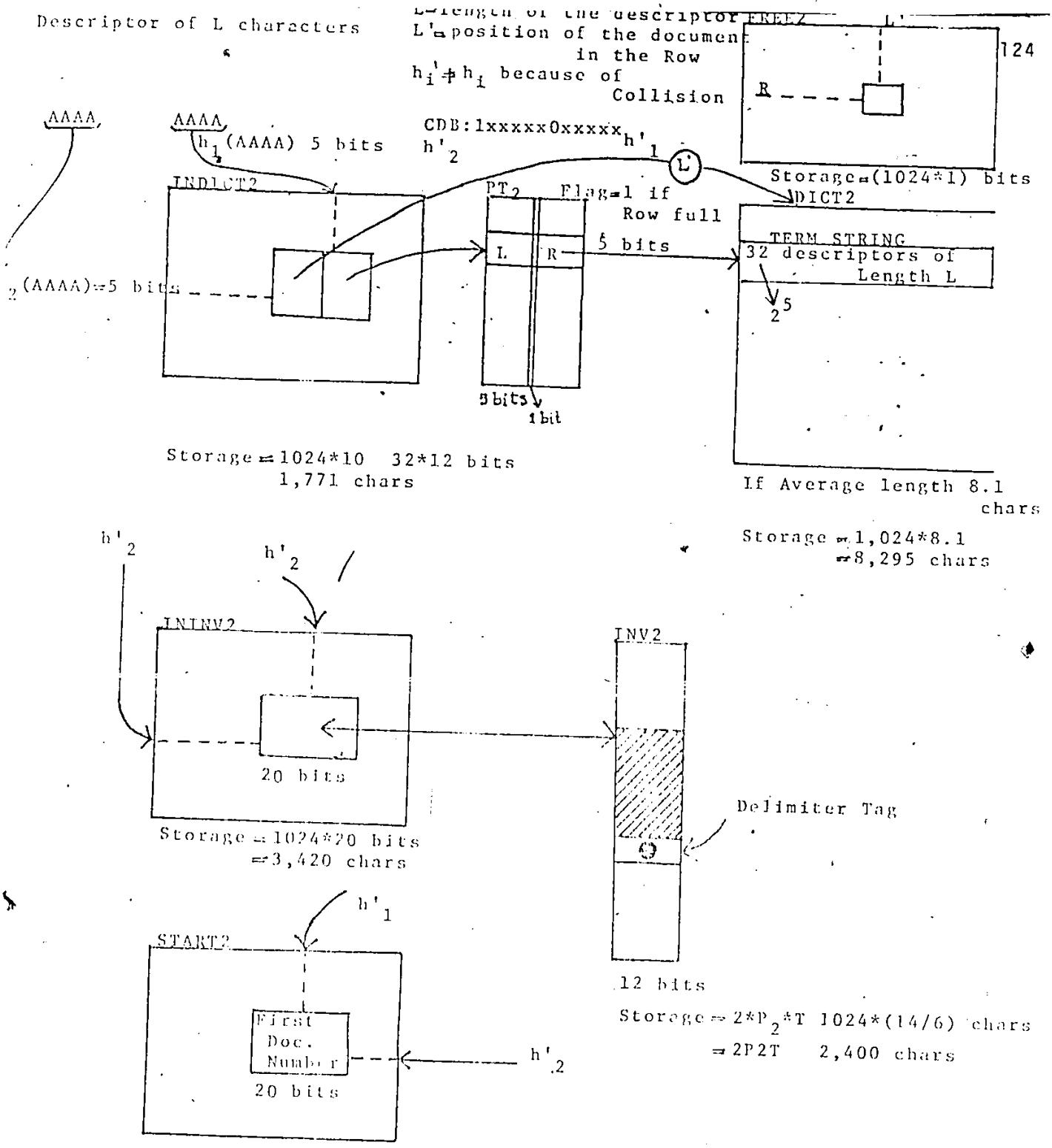


Fig. 7

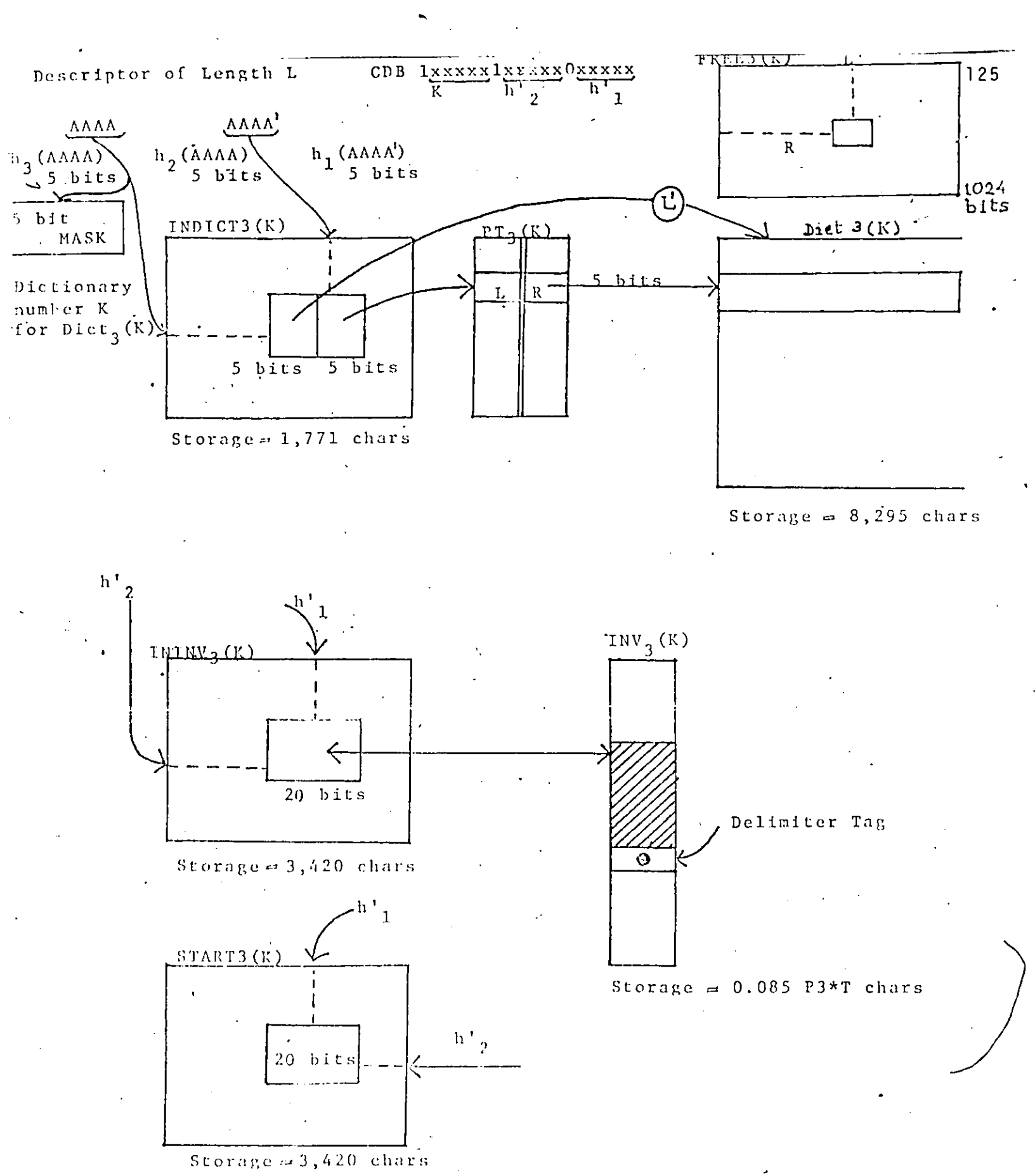

Descriptor of Length L
CDB 1xxxxx1xxxxx0xxxxx
K
h'2
h'1
AAAA
AAAA'
h3(AAAA) 5 bits
h2(AAAA) 5 bits
h1(AAAA') 5 bits
5 bit MASK
Dictionary number K for Dict3(K)
INDICT3(K)
5 bits 5 bits
Storage = 1,771 chars
PT3(K)
L
R
5 bits
L'
FREE3(K)
125
R
1024 bits
Dict 3(K)
Storage = 8,295 chars
h'2
h'1
ININV3(K)
20 bits
Storage = 3,420 chars
INV3(K)
Delimiter Tag
Storage = 0.085 P3*T chars
h'1
START3(K)
20 bits
h'2
Storage = 3,420 chars

Fig. 8

POOR COPY

APPENDIX A

LISTS OF THE 13 SETS OF DATA.

List - 1

Home, small, found, mrs, thought, went, say, part, once, general, high, A, upon, school, every, don't, does, got, united, left, number, course, war, until, always, away, something, B, fact, though, water, less.

List - 2

Both, might, old, since, another, while, being, come, day, against, year, between, never, under, off, here, know, last, great, US, same, get, life, go, came, right, used, taxe, three, states, himself, few.

List - 3

A, he, I, are, was, had, of, at, be, as, by, it, not, is, the, on, for, in, with, and, this, that, to, his, but, from, or, have, an, they, which, one.

List - 4

If, new, about, out, only, can, then, its, will, when, what, we, other, into, who, said, has, up, than, him, more, been, so, no, some, could, time, these, two, may, then, do.

List - 5

Way, many, me, even, where, like, did, your, much, through, well, before, man, our, back, such, years, also, must, after, most, now, made, over, down, should, because, each, just, those, people, mr.

POOR COPY

List - 6

Ago, became, special, problems, free, behind, brought, air, whose, real, political, office, miss, probably, cannot, sure, major, seems, held, making, keep, heard, question, period, itself, it's, york, times, human, law, line, above.

List - 7

Study, a, education, new, of, state, law, history, from, world, american, development, an, the, life, on, guide, for, states, in, with, and, report, to, united, by, at, economic, book, america, modern, english.

List - 8

A, b, c, of, structure, acid, poly, synthesis, methyl, from, oxide, the, compounds, on, determination, for, to, in, with, and, effect, by, di, an, d, properties, tri, reaction, amino, acids, derivatives, carbon.

List - 9

Night, knew, later, eyes, city, didn't, point, look, business, asked, government, why, going, nothing, told, better, program, system, find, next, end, called, set, give, group, toward, young, days, let, room, president, said.

List - 10

Whether, week, really, c, five, taken, feet, history, anything, body, experience, seen, past, half, show, gave, having, either, act, quite, today, local, death, accross, car, field, word, words, already, themselves, information, am.

List - 11

Areas, change, board, individual, west, tomato, community, south, love, society, job, turn, close, zero, into, network, un, deux, trois, quatre, cinq, six, sept, huit, tony, er, tu, guerre, famille, tout, vingt, neuf.

POOR COPY

List - 12

Wife, age, amino, future, papa, et, department, wanted, full, oui, voice, red, force, guerre, seem, mama, un, true, court, cost.

List - 13

Things, ever, best, power, need, felt, case, children, big, church, rather, along, least, early, four, among, light, within, development, become, saw, John, form, large, looked, per, often, white, important, second, possible, face.

POOR COPY]

APPENDIX B

List of cls

A (AT, AS, AN, AM, A)
AC
AG (AGO, AGE)
AI
AN
AR
ACI
ALS
AWA
ABOU
ABOV
ACCR
ACID
AGAI
AFTE
ALON
ALRE
ALWA
AMER
AMIN
AMON
ANOT
ANYT
AREA
ASKE
B (BE, BY)
BI
BU
BAC
BEE
BES
BOD
BOO
BOT
BECA (BECAME, BECAUSE)
BECO
BEFO
BEHI
BEIN
BETT
BETW
BOAR
BROU
BUSI
C
CA
CAM
CAS
CIN
CIT
COM
COS
CALL
CANN
CARB
CHAN
CHIL
CHUR
CLOS
COMM
COMP
COUL
COUR
D (DO, DI, D)
DA
DI
DAY
DEU
DOE
DOW
DEAT
DEPA
DERI
DETE
DEVE
DIDN
DON'
E (ET, ER)
EN
EAC
EVE
EYE
EARL
ECON
EDUC
EFFE
EITH
ENGL
EVER
EXPE
FE
FO
FAC
FEE
FEL
FIN
FIV
FOR
FOU

POOR COPY]

FRE
FRO
FUL
FAMI
FIEL
FOUN
FORC
FUTU
G
GE
GO
GAV
GIV
GENE
GOIN
GOVE
GREA
GROU
GUER
GUID
H
HA (HAS, HAD)
HE (HIM, HIS)
HAL
HAV
HEL
HER
HIG
HOL
HOU
HAVI
HEAR
HIMS
HIST
HUMA
I (IT, IS, IN, IF, I)
IT
INT
IT
INDI
INFO
IMPO
ITSE
JO
JOH
JUS
KEE
KNE
KNO
LA
LE
LAS
LEF
LES
LIF
LIN
LIK
LOO
LOV
LARG
LATE
LEAS
LIGH
LOCA
LOOK
M (ME, MR)
MA (MAY, MAN)
MR
MAD
MAM
MAN
MIS
MOR
MOS
MUC
MUS
MAJO
MAKI
METH
MIGH
MODE
N
NE
NO (NOW, NOT)
NEE
NEU
NEX
NETW
NEVE
NIGH
NOTH
NUMB
O (OF, ON, OR)
OF
OL
ON
OU (OUI, OUT, OUR)
ONC
ONL
OVE
OFFI
OFTE
OTHE
OXID
PE
PAP
PAR
PAS
POL

POOR COPY

PEOP
PERI
POIN
POLI
POSS
POWE
PRES
PROB (PROBLEMS, PROBABLY)
PROG
PROP
QUAT
QUES
QUIT
RE
REA
ROO
RATH
REAC
REAL
REPO
RIGH
S
SA (SAY, SAW)
SE
SI
SAL
SAM
SEE
SEP
SHO
SOM
SUC
SUR
SECO
SCHO

SEEM
SHOU
SINC
SMAL
SOCI
SOUT
SOME
SPEC
STAT (STATE, STATES)
STRU
STUD
SYNT
SYST
T (TO, TU)
TH
TR
TW
TAX
THA (THAT, THAN)
THE (THEY, THEN)
THI
TIM
TOL
TON
TOU
TRU
TUR
TAKE
THEN
THES
THIN
THOS
THOU (THOUGH, THOUGHT)
THRE
THRO
TIME

TODA
TOMA
TOWA
TROI
U (UP, US, UN)
UPO
USE
UNDE
UNIT
UNTI
VIGN
VOIC
W
WA (WAR, WAS, WAY)
WH (WHY, WHO)
WEE
WEL
WEN
WLS
WHA
WHE
WIF
WIL
WIT
WOR
WANT
WATE
WHER
WHET
WHIC
WHIL
WHIT
WHOS
WITH
WORL
WORD

YEA
YOR
YOU
YEAR
YOUN
ZER

POOR COPY

APPENDIX C

Lists of c2s

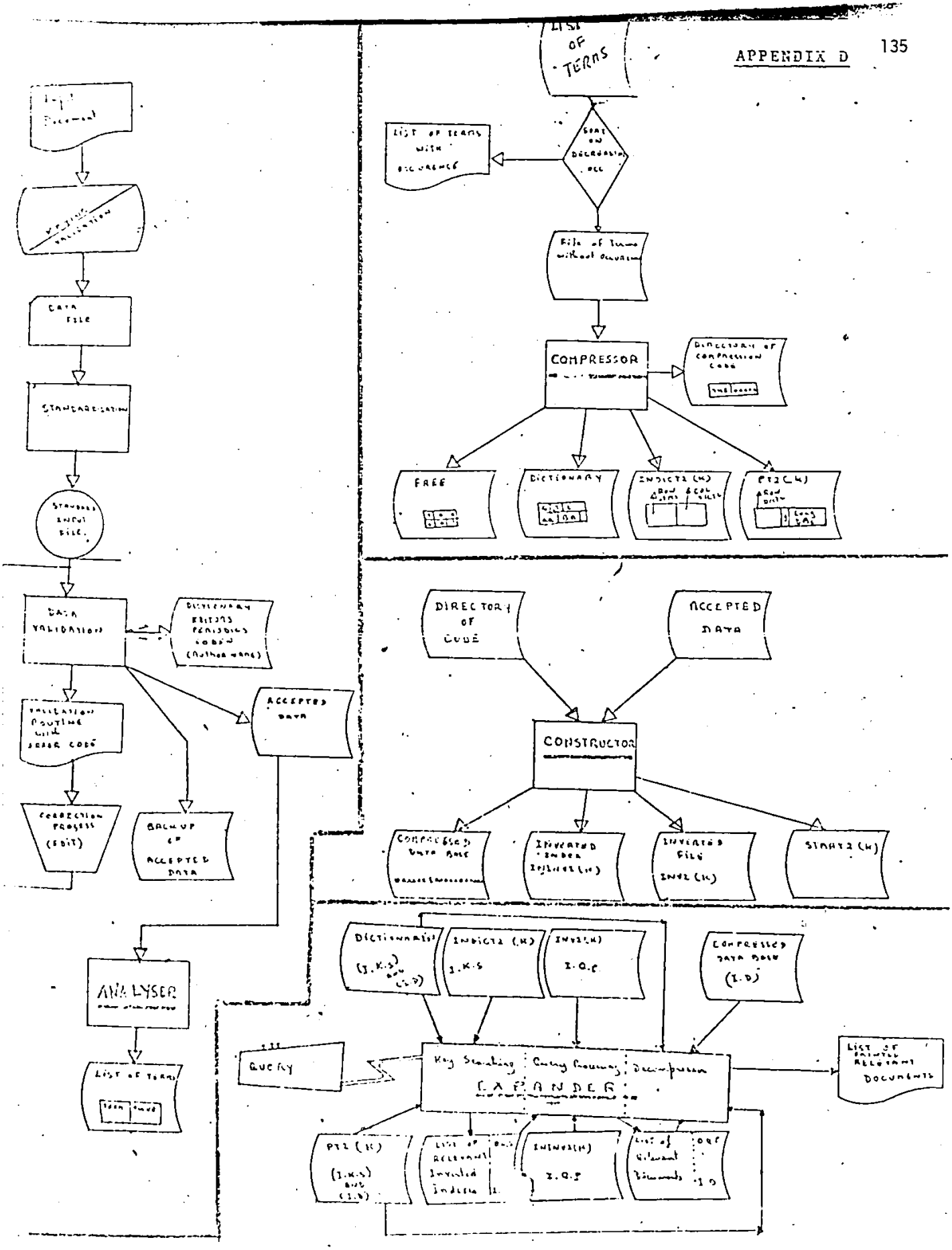
A
AD
AN (MAN, CAN)
AS (AS, WAS)
AR (CAR, WAR)
AW (LAW, SAW)
AY (SAY, BAY, MAY, WAY)
ACE
ACH
ACK
ACT
ADE
AID
ALF
AMA
AME (CAME, SAME)
ANY
APA
ART
ASE
AST (LAST, PAST)
AVE (HAVE, GAVE)
AXE
AYS
ABLY
AJOR
AKEN
ALLY
ANGE
ARGE
ARLY
ATER
ATES
ATRE
AUSE
B
BOUT
BOVE
C
CT
CID
CAME
CIAL
CIDS
COME
COND
D
DENT
DERN
DN'T
DREW
DUAL
E (HE, BE, WE, ME)
ED
ER
ET (SET, GET, LET)
EW (NEW, FEW)
EAL
EAR
EED
EEK
EEM
EEN (SEEN, BEEN)
EEP
EET
EFT
ELD
ELL
ELT
ENT
EPT
ERE
ERN
ERO
ESS
EST (BEST, WEST)
EUF
EUX
EXT
EADY
EARD
EARS
EAST
EATH
EEMS
EING
ENCE
ERRE
ERAL
ESIS
EVER
F (IF, OF)
FF
FECT
FICE
FORE
FTEN
FTER

POOR COPY]

GE
GO
GRAN
HE
HO
HY
HAN
HAT (THAT, WHAT)
HEN (THEN, WHEN)
HEY
HIS
HOW
HERE
HESE
HICH
HILE
HIND
HING (ANYTHING, NOTHING, SOMETHING)
HITE
HOOL
HOSE (WHOSE, THOSE)
HREE
I (I, DI)
ID
IC
IM
IR
IS
IX
IFE (LIFE, WIFE)
ICH
IKE
ILL
ITH
IME
IND
INE
INQ
ISS
ITY
IVE (FIVE, GIVE)
ICAL
IBLE
ICAN
IELD
IETY
IGHT (LIGHT, NIGHT, RIGHT, MIGHT)
ILLE
IMES
INCE
INGS
INGT
INST
ITED
IVES
KING
LD
LSO
LEMS
LISH
LLED
LONG
LOSE
LVES-
M
MALL
MATO
MBER
MENT
MINO
MONG
N (ON, IN, AN, UN)
ND (AND, END)
NE
NCE
NEW
NLY
NOW
NTO
NDER
NESS
NITY
NNOT
NTED
NTIL
O (TO, SO, NO, DO, GO)
OL
OR
OT (NOT, GOT)
OW
ODY
OES
OHN
OLD
OLY
OME (HOME, COME, SOME)
ONY
OOK (LOOK, BOOK)
OOM
ORD
ORE
ORK
ORN

POOR COPY]

OST (COST, MOST)
OTH
OUR
OUT
OVE
OWN
OARD
OCAL
ODAY
OICE
OING
OINT
OKED
OMIC
ON'T
OPLE
ORCE
ORDS
ORLD
OUGH (THOUGH, THROUGH)
OULD (SHOULD, COULD)
OUND
OUNG
OURT
OUTH
OWER
P
PQN
PORT
PHENT
R (MR, OR, ER)
RE
RI
RS
REE
ROM
RUE
RBON
REAS
REAT
RICA
RIOD
ROIS
ROUP
ROSS
S (AS, US, IS)
SED
SELF (HIMSELF, ITSELF)
SKED
STEM
T (IT, AT, ET)
TS
T'S
TANT
TATE
THER (RATHER, OTHER, NETHER, EITHER, ANOTHER)
THIN
THYL
TIES
TION (QUESTION, EDUCATION, DETERMINATION, REACTION, INFORMATION)
TORY
TTER
TUDY
TURE
U
UI
UR
UT (OUT, BUT)
UCH (MUCH, SUCH)
UIT
ULL
URE
URN
UST (JUST, MUST)
UGHT (BROUGHT, THOUGHT)
UIDE
UITE
UMAN
UNDS
URCH
URSE
VEN
VER
VERY
VING
WO
WAY
WARD
WAYS
WEEN
WORK
XIDE
Y
YES


LIST OF TERMS
SORT ON DECREASING FREQ
LIST OF TERMS WITH FREQUENCY
File of Terms without frequency
COMPRESSOR
Directory of compression code
FREE
DICTIONARY
INDEX1 (M)
PTR (M)
DATA FILE
STANDARDISATION
DATA VALIDATION
ACCEPTED DATA
CORRECTION PROCESS (EDIT)
BACKUP OF ACCEPTED DATA
DIRECTORY OF CODE
ACCEPTED DATA
CONSTRUCTOR
COMPRESSED DATA BASE
INVERTED INDEX
INVERTED FILE
START1 (M)
ANALYSER
LIST OF TERMS
QUERY
Key Stacking
Query Processing
Decompression
EXPANDER
LIST OF RELEVANT DOCUMENTS
PTR (M)
INDEX1(M)