File Structure for an On-Line Catalog of One Million Titles

J. J. DIMSDALE: Department of Computing Science, University of Alberta, Edmonton, Canada, and H. S. HEAPS: Department of Computer Science, Sir George Williams University, Montreal, Canada.

A description is given of the file organization and design of an on-line catalog suitable for automation of a library of one million books. A method of virtual hash addressing allows rapid search of the indexes to the catalog file. Storage of textual material in a compressed form allows considerable reduction in storage costs.

INTRODUCTION

An integrated system for on-line library automation requires a number of computer accessible files. It proves convenient to divide these files into three principal groups, those required for the on-line catalog subsystem, those required for the acquisition subsystem, and those required for the on-line circulation subsystem. The present paper is concerned with the files for the catalog subsystem. Files required for the circulation subsystem will be discussed in a future paper.

The files for an on-line catalog system should contain all bibliographic details normally present in a manual catalog, and the file should be organized to allow searches to be made with respect to title words, authors, and Library of Congress (LC) call numbers. It may also be desired to search on other bibliographic details, in which instance the appropriate files may be added to those described in the present paper.

The file organization should be such as to support economic searching with respect to questions in which terms are connected by the logic operations AND, OR, and NOT. It should also allow question terms to be connected by operations of ADJACENCY and PRECEDENCE, and it should allow question terms to be weighted and the search made with reference to a specified threshold weight. It may be desirable for the file organization to include a thesaurus that may be used either directly by the user or by the search program to narrow, or broaden, the scope of the initial query or to ensure standardization of the question vocabulary.

The file organization and search strategy should ensure that the user of the on-line catalog system receive an acceptable response time to his queries, although it is likely that some of the operations required by the circulation system will be given a higher priority. Thus the integrated system must time-share between search queries, circulation transactions, and other tasks that originate from a number of separate terminals or from batch input. Such tasks might arise from acquisitions, and from update and maintenance of the on-line catalog. The system should be a special purpose time-sharing system such as the Time Sharing Chemical Information Retrieval System described by Lefkovitz and Powers and by Weinberg.[1, 2] In this system the queries time-share disk storage as well as the central processor.

Since an on-line catalog is a large file, and hence expensive to store in computer accessible form, it is desirable to store it in as compact a form as possible. For example, a catalog file for one million titles is likely to involve between 2×10^8 and 5×10^8 alphanumeric characters. If stored character by character the required storage capacity would be equivalent to that supplied by from seven to sixteen IBM 2316 disk packs. It is also important to design the frequently accessed files so as to minimize the number of disk, or data cell, accesses required to process each query.

The files described in the present paper include ones stored in compressed form and organized for rapid access.

Throughout the present paper the term title is used in a general sense. It may include periodical titles as well as book titles. However, it is supposed that frequently changing information, such as periodical volume ranges, will be stored as part of the circulation subsystem rather than the catalog subsystem.

OVERALL FILE ORGANIZATION

The complete bibliographic entries of the catalog may be stored in a serial (sequential) file so that any record may readily be read and displayed in its entirety. However, as indicated by Curtice, use of an inverted file is to be preferred for purposes of searching.[3] An alternative to the simple serial file is one organized in the form of a multiple threaded list (multilist) in which all records that contain a particular key are linked together by pointers within the records themselves. The first record in each list is pointed to by an entry in a key directory as described by Lefkovitz, Holbrook, Dodd, and Rettenmayer.[4–7]

For very small collections of documents Divett and Burnaugh have attempted to organize on-line catalogs by use of ring structured variations of the multilist technique.[8, 9] Neither file organization is feasible for a collection of a million documents because of the long length of the threads involved. Many disk accesses would be needed in order to retrieve all elements of a list, and hence there would be a very slow response to queries. The cellular multilist structure proposed by Lefkovitz and Powers, or the cellular serial structure proposed by Lefkovitz, may well prove to be a viable alternative to the organization proposed in the present paper.[10, 11]

However, as indicated by Lefkovitz, the inverted organization provides shorter initial, and successive, response times in answer to queries.[12]

In the present paper it is supposed that 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. Such a two-level structure is often assumed and has been termed a "combined file" by Warheit who concluded it to be superior to either a single serial file or a threaded list organization.[13-17]

The file structure described in the present paper uses indexes based on the virtual scatter table as described by Morris and Murray, the scatter index table discussed by Morris, and the bucket as treated by Buchholz.[18-20] The attractiveness of a similar structure for use in the Ohio College Library Center has been analyzed by Long, et al.[21] The basic elements of the file organization are shown in Figure 1. It is supposed that the access keys are title words, but a similar file structure is used for access with respect to keys of other types.

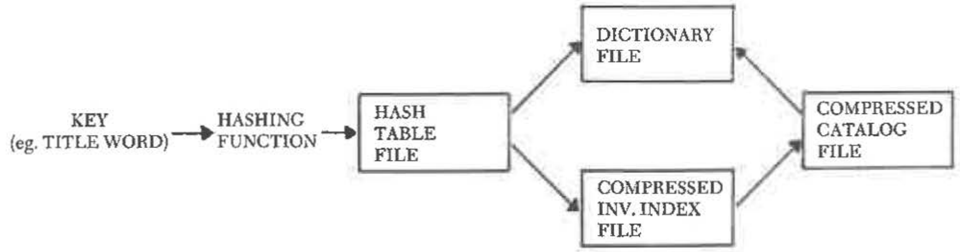

Fig. 1. Overall File Organization

Any 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 which is stored in a compressed form. Entries within the compressed inverted index serve as pointers to the catalog file of complete bibliographic entries. Terms, such as title words, within the catalog file are coded to allow a compressed form of storage. The codes used in the compressed catalog file serve as pointers to the uncoded terms stored in the dictionary file.

There would be a separate hashing function, hash table file, dictionary file, and compressed inverted file for use with each different type of key. However, there is only one compressed catalog file.

For a search scheme that allows use of a thesaurus of synonyms, narrower terms, broader terms, and so forth, a thesaurus file may be added (Figure 2).

The files must be organized to allow for ease of updating. As further bibliographic entries are added it is necessary to add additional pointers from the inverted index. Also, whenever a new key occurs in a bibliographic entry it must be added to the dictionary, assigned a code for storage in the compressed catalog file, and entered into the compressed inverted index.

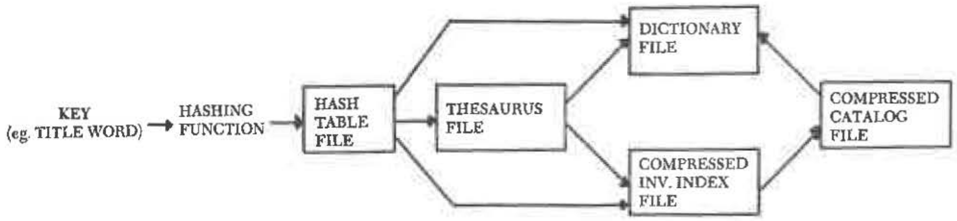

Fig. 2. File Organization with Inclusion of a Thesaurus

STRUCTURE OF THE HASH TABLE FILE

In order to locate the set of inverted index pointers that corresponds to a given search key K, the key is first operated on by a hashing function that transforms it into a bit string of length v bits. Each such bit string is said to represent a virtual hash address, and is regarded as the concatenation of two substrings of length r and v-r bits. The two substrings are respectively said to constitute the major and the minor M(K) of the virtual hash address. The major is further divided into two bit strings B(K) and I(K) that define a bucket number B(K) of a bucket β(K), and an index number I(K) of an entry within the bucket. The major that represents the pair of numbers B(K), I(K) is said to constitute a real hash address.

The hash table file is divided into portions, or buckets, of equal length. Each bucket is further divided into an index section, a content section, and a counter section (Figure 3). The index sections of all buckets have the same length. Similarly, all content sections are of equal length, and so are all counter sections.

As the hash table is created, entries are added sequentially into the content section so that any unfilled portion is at the end. In contrast, the index section of any bucket may contain unfilled entries at random positions and hence constitutes a scatter table.

The hash table file is created as follows. The various keys are transformed by the hashing function into bit strings B(K), I(K), M(K). In the bucket β(K) of number B(K) an entry as described below is added to the content section, and the vacancy pointer within the counter section is incremented to point to the beginning of the unfilled portion of the content section. The I(K)th entry number in the index section is then set to point to the position of the entry added to the content section. The entry placed in the content section includes the minor M(K) and a dictionary pointer to where the key is placed in the dictionary file as well as a pointer to an entry in the compressed inverted index.

If there has previously occurred a bit string $B(K_1)$, $I(K_1)$, $M(K_1)$ in which $B(K_1) = B(K)$, $I(K_1) = I(K)$, $M(K_1) \neq M(K)$ then no change is

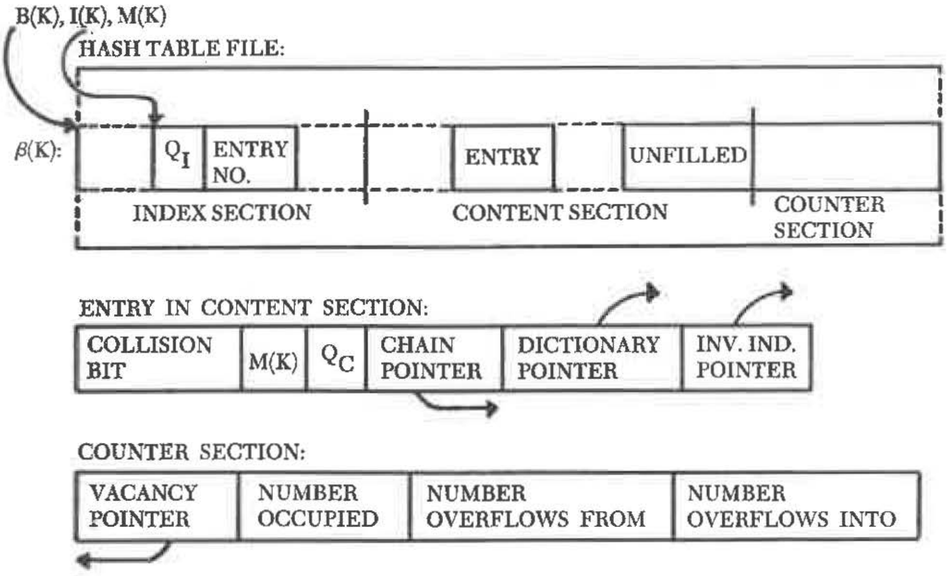

Fig. 3. Bucket of the Hash Table File

made to the I(K)th entry in the bucket $\beta(K)$ or to the minor $M(K_1)$ in the content section. Instead, the chain pointer is set to point to the location of a new entry that is added to the content section. In this new entry the minor is set to M(K) and the dictionary pointer is set to indicate where the new key is placed in the dictionary file. There is said to have resulted a collision at the real hash address B(K), I(K).

If there has previously occurred a bit string $B(K_1)$, $I(K_1)$, $M(K_1)$ in which $B(K_1) = B(K)$, $I(K_1) = I(K)$, $M(K_1) = M(K)$, where $K_1 \neq K$, then the collision bit that precedes $M(K_1)$ is set to 1. and a further content entry containing M(K) is chained from the entry that contains $M(K_1)$. There is said to have occurred a collision at the virtual hash address B(K), I(K), M(K).

The last three entries included in the counter section shown in Figure 3 are optional but are useful for monitoring the performance of the hashing function with respect to bucket overflows and so forth.

A bucket becomes full when there is no remaining unfilled space in its content section. If a further chain pointer is required from a content entry, its preceding overflow bit Q_C is set to 1 to indicate that the pointer is to another bucket. Likewise, if a further entry is required in the index section its preceding overflow bit Q_I is set to 1 to indicate that it refers to an entry within another bucket. The bucket is then said to have overflowed. Methods of handling bucket overflow, and choice of the new bucket, are discussed in a subsequent section.

It should be noted that use of a hash table as described above retains most of the advantages of the usual scatter index method in which the index entries and content entries are stored in two separate files. It has the further important advantage that in most instances a single disk access is sufficient to locate both the index entry and the corresponding content entry.

As noted by Buchholz and Heising, if it is known that certain keys are likely to appear with high frequency in search queries then it is advantageous to enter them at the start of creation of the hash file.[22, 23] They will then tend to appear near the beginnings of the content entry chains and hence require little CPU time for their subsequent location. Furthermore, they will tend to appear in the same bucket as their corresponding index entries, and hence their location will usually require only a single disk access.

NUMBER OF BITS FOR VIRTUAL HASH ADDRESS

Suppose the hashing function is chosen so that the majors of the transformed keys are uniformly distributed among the R slots available for real hash addresses B,I. If there are N keys then $\alpha = N/R$ may be termed the load factor. It is the average number of keys that are transformed into any given real hash address.

The probability that any given real hash address corresponds to k keys is given by Murray[24] as

$$P_k = e^{-\alpha}\,\alpha^k/k! \tag{1}$$

Hence, for any given real address the probability of a collision occurring is

$$C = \sum_{k=2}^{N} P_k = 1 - P_0 - P_1 = 1 - (1+\alpha)e^{-\alpha}. \tag{2}$$

If a collision occurs at a particular real hash address, the expected length of the required chain within the content section is

$$\begin{aligned} L &= \sum_{k=2}^{N} kP_k/C \\ &= (1/C)\,\Big(\sum_{k=0}^{N} kP_k - P_1\Big) \\ &= (1/C)\,(\alpha - \alpha e^{-\alpha}) \\ &= \frac{\alpha\,(e^{\alpha} - 1)}{e^{\alpha} - 1 - \alpha} \end{aligned} \tag{3}$$

It may be noted that if the load factor α is equal to 1 then $L = 2.43$.

If all the transformed keys are distributed uniformly among the V possible virtual addresses B, I, M then the expected total number of collisions at virtual addresses is given by Murray[25] as

$$\overline{p} = N^2/2V \tag{4}$$

provided $V \gg N$. The expected relative frequency of collisions at virtual addresses is therefore

$$f = N/2V. \tag{5}$$

It proves convenient to regard N, f, and α as basic parameters in terms of which may be determined the number r of bits required in the major, and the number v of bits required in the virtual hash addresses.

The value of r must be at least as large as $\log_2 R = \log_2(N/\alpha)$, and hence r may be chosen according to the formula

$$r = \ulcorner \log_2 (N/\alpha) \tag{6}$$

where $\ulcorner$ means "the smallest integer greater than or equal to." The value of v must be at least as large as

$$v = \ulcorner \log_2 V = \ulcorner \log_2 (N/2f). \tag{7}$$

If N and f have the form $N = 2^n$ and $f = 2^{-\gamma}$ then v may be chosen according to the formula

$$v = n + \gamma - 1 \tag{8}$$

and the number of bits required for the minor is

$$m = v - r. \tag{9}$$

CHOICE OF BUCKET CAPACITY

With an 8-bit byte-oriented computer, such as the IBM 360, it proves convenient to use 8 bits of storage for each entry number plus overflow bit within the index section. If a value of zero is used to indicate an unused index entry there remain up to 127 possible values for entry numbers. Thus the number c of entries in the content section must be less than or equal to 127.

Suppose there are b slots for index entries in each bucket. The total number of index entries in the entire file is R. It follows from the results of Schay and Spruth,[26] Tainter,[27] and Heising[28] that the probability P(b, c) of overflow of any bucket is given by

$$P(b,c) = \sum_{k=c+1}^{\infty} (\alpha b)^k \frac{c^{-\alpha b}}{k!}. \tag{10}$$

For selected values of b, Beyer's tables of the Poisson distribution have been used to compute P(b, c) and to determine the largest value of c for which $P(b, c) \leq 0.01$.[29] The results are shown in Table 1 for the instance in which $\alpha = 1$. A similar table has been computed by Buchholz[30] for the instance in which $c = b$ and α ranges from 0.1 to 1.2.

As is apparent from Table 1, an increase in the value of b allows use of a smaller ratio c/b and hence permits more economical use of storage. With $b = 64$ the allowed value of c/b is 1.33 and hence c may be chosen equal to 85.

The reduction in access time that results from structuring the file so that each bucket contains both index and content entries is, of course, effected at the expense of additional storage costs. For example, if $c/b = 1.33$ then the space allocated for storage of content entries is 33 percent greater than if content entries are stored in a separate file. Relaxation of the condition $P(b,c) \leq 0.01$ allows a reduction in c/b, but the increased number of bucket overflows will cause additional disk accesses to be required.

Table 1. Values of b, c, and c/b for which P(b,c)≤0.01 when α = 1.

b	*c*	*c/b*
1	5	5.00
2	6	3.00
3	8	2.66
4	10	2.50
5	11	2.20
6	13	2.17
7	14	2.00
8	15	1.88
9	17	1.89
10	18	1.80
11	19	1.73
12	20	1.67
13	22	1.69
14	23	1.64
15	24	1.60
16	25	1.56
17	27	1.59
18	28	1.55
19	29	1.53
20	30	1.50
60	80	1.33
100	125	1.25

TREATMENT OF BUCKET OVERFLOWS

When a new key is found to map into a bucket whose content section is full then some means must be found to provide space in some other bucket. The particular procedure that should be used depends on the extent to which the entire set of buckets contain unfilled portions.

Suppose that many buckets are almost full and that the number c of allowed content entries is less than 127. The entire hash file may then be expanded with the same index sections but with longer content sections.

If many buckets are almost full and c = 127 then the entire file may be expanded in such manner that each bucket is replaced by a pair of buckets that contain the same number b of allowable index entries, but whose number c_1 of allowable content entries is chosen to ensure that $P(b,c_1) \leq 0.01$. Such doubling of buckets also doubles the number of index entries but it does not double the storage required for the entire file. Each key K that corresponds to an entry in the original bucket is associated with an entry in the first, or second, of the new buckets according as the leading bit of either its index address I(K) or its minor M(K) is equal to 0 or 1. The effect is to shift one bit from I(K) or M(K) into the bucket address B(K). This method is based on a suggestion of Morris.[31]

Suppose that few buckets are almost full. Then a suitable means of determination of an unfilled bucket for storage of the minor is through use of some overflow algorithm that determines a sequence of bucket numbers $B_0(K)$, $B_1(K)$, $B_2(K)$, etc., corresponding to any given full bucket $\beta_0(K)$. Suppose there are n_b buckets. A quadratic residue algorithm

(11) $$B_j(K) = [B_0(K) + aj + bj^2] \bmod n_b$$

has been considered by Maurer and by Bell for use with in-core hash tables, but it suffers from the disadvantage that the existence of a full bucket $\beta_0(K)$ will divert entries into the particular buckets $\beta_1(K)$, $\beta_2(K)$, etc. and hence cause them to fill more rapidly than other buckets which may contain fewer entries.[32, 33]

It is believed that a more desirable form of the quadratic residue algorithm is

(12) $$B_j(K) = \{B_0(K) + f_j[I(K)]\} \bmod n_b$$

where f_j is a suitably chosen function. Letting $B_j(K)$ depend, through f_j, on both j and I(K), instead of on j alone, allows reduction of the tendency to fill a particular set of buckets.

To prevent a tendency to overflow particular buckets it is also desirable for the overflow algorithm to produce bucket numbers that are uniformly distributed among all possible bucket numbers. Among the more promising forms to be chosen for the $f_j[I(K)]$ are the following

(13a) $$f_j[I(K)] = I'(K)\,j$$

where $j = 1, 2, \ldots, n_b - 1$, and $I'(K)$ denotes I(K) if I(K) is odd, but denotes $I(K) + 1$ if I(K) is even. Since n_b is a power of 2 such choice of $I'(K)$ ensures that $I'(K)$ and n_b have no common factors, and hence that $B_j(K)$ steps through the sequence $\beta_0(K)$, $\beta_1(K)$, etc. covering every bucket in the file.

(13b) $$f_j[I(K)] = I'(K)\,j^2$$

where $j = 1, 2, \ldots, \ulcorner\sqrt{n}-1$, and $\ulcorner$ means "the least integer greater than or equal to."

(13c) $$f_j[I(K)] = R_j[I'(K)]$$

where $j = 1, 2, \ldots, n_b$, and $R_j[I'(K)]$ denotes a number output by a pseudorandom number generator of the form suggested by Morris[34] with an initial input of $I'(K)$ instead of 1.

It may be remarked that use of Equation 13a requires the least number of machine instructions, and the least CPU time per step, but it has a strong tendency to cluster the $\beta_j(K)$ immediately after the $\beta_0(K)$ and hence it is likely to be the least effective of the three methods. Use of Equation 13b produces less clustering, but the sequence does not include all buckets of the file. Use of Equation 13c requires the largest number of instructions and CPU time per step, but the $\beta_j(K)$ are less likely to cluster and they are uniformly distributed among all possible buckets. Thus Equation 13c produces shorter chains of overflow buckets and hence requires fewer disk accesses.

If a new key K maps into a full bucket $\beta_0(K)$ then the following procedure is used to determine the bucket into which the minor of K is to be inserted:

(i) The chain of pointers from the I(K)th entry of the bucket $\beta_0(K)$ is followed, possibly through overflow buckets given by Equation 12, in order to locate the terminal entry of the chain. Suppose this terminal entry is within a bucket $\beta_j(K)$.

(ii) If there is available space in bucket $\beta_j(K)$ then the minor $M(K)$ is entered and chained as described previously.

(iii) If bucket $\beta_j(K)$ is full, but there is space in $\beta_{j+1}(K)$, then the minor $M(K)$ is entered into $\beta_{j+1}(K)$ and chained as described previously.

(iv) If buckets $\beta_j(K)$ and $\beta_{j+1}(K)$ are both full, and bucket $\beta_{j+1}(K)$ contains at least one nonempty index entry $I(K')$ whose chained content entries are all contained within $\beta_{j+1}(K)$, then the minor $M(K)$ is stored according to the following displacement algorithm:

The terminal member of the chain from $I(K')$ is displaced to an overflow bucket $\beta_r(K')$ determined by use of Equation 12, except that if both $\beta_r(K')$ and $\beta_{r+1}(K')$ are full then a further bucket is determined by use of the displacement algorithm. The minor $M(K)$ is substituted for the displaced entry in bucket $\beta_{j+1}(K)$ and is chained appropriately.

(v) If application of Step (iv) leads to a bucket $\beta_{j+1}(K)$, or $\beta_{r+1}(K)$, that contains no nonempty index entry whose chained content entries are all contained within it, then the entire hash file must be expanded by use of one of the procedures described at the beginning of the present section.

It should be emphasized that, although Step (iv) is necessary for completeness, the probability of its use is very low. With a probability of less than 0.01 for a bucket overflow, the probability of use of Step (iv) is less than $(0.01)^3$.

SEARCH PHASE AND PROBLEM OF MISMATCH

In the previous sections the structure of the hash index file has been discussed with emphasis on details of its creation and update. During search of the catalog files by use of the inverted index, each search key is processed by the following search alogorithm:

Step 1: The search key K is transformed by the hashing function into a virtual hash address $B(K), I(K), M(K)$.

Step 2: The bucket $\beta(K)$ is read into core.

Step 3: The index entry specified by $I(K)$ is examined. If it is empty then the search key is not present in the data base. If it is not empty then Step 4 is performed.

Step 4: The overflow bit of the index entry specified by $I(K)$ is examined. If it is equal to 1 then Step 5 is performed. If it is equal to 0 then Step 6 is performed.

Step 5: The overflow algorithm is used to determine the address of the required overflow bucket which is then read into core, and Step 6 is executed.

Step 6: The minor of each entry in the chain of content entries is compared to the minor of the search key's virtual hash address until either a match is found or the chain is exhausted. Whenever the chain leads to an overflow bucket then Step 5 is performed.

Step 7: If a match is found for M(K) then the collision bit of the entry is examined. If it is equal to 0 then Step 9 is performed. If it is equal to 1 then Step 8 is performed.

Step 8: The dictionary entry that corresponds to each content entry in the virtual address collision is read into core and compared to the search key K. If no match is found then the search key is not present in the index.

Step 9: This step is included because there is a small probability that a misspelled search key, or one not present in the hash file, may be transformed into the same virtual address as some key already included in the file. The step consists of reading the corresponding dictionary entry into core and comparing it with the search key. For reasons discussed later in the present section it is desirable to omit this step.

It should be noted that in most instances the search algorithm will not require execution of Steps 5 and 8. In fact, with the hash index files designed as described in the previous sections, the probability of execution of Step 5 is about 0.01 and the probability of execution of Step 8 is about 2^{-16}. Consequently, if Step 9 is also omitted the number of disk accesses required to find the index entry corresponding to a search key is approximately 1.01.

The mismatch problem, which gives rise to Step 9 of the search algorithm, is less serious than might be expected. Suppose the hash function distributes the transformed keys uniformly over all hash addresses. The probability that a new, or misspelled, key maps into an existing entry is given by

(14) $$P_c = N/V$$

The probability that a search leads to a mismatch is therefore

(15) $$P_m = P_s N/V$$

where P_s is the probability that the search key is misspelled or not in the hash table. Thus, for a hash table of $N = 2^{16} = 65{,}536$ title words and $V = 2^{31}$, an assumption of $P_s = 0.1$ leads to $P_m = 3 \times 10^{-6}$.

Because P_m is extremely small, and because each execution of Step 9 requires up to two disk accesses, it is desirable to omit this step. If experience shows that particular new or misspelled search keys occur frequently, and cause mismatches, they may themselves be entered into the hash index file. In fact, some degree of automatic spelling correction may be provided if some common misspellings are included in the hash files and chained to the content entries that correspond to the correctly spelled keys. Correct, but alternative, spellings of search keys may also be treated in the same manner.

SIZE OF HASH FILE FOR TITLE WORDS

Suppose the document collection contains T different titles that comprise a total of W words of which there are N different words. Let $\overline{W} = W/T$ denote the average number of words in each title. Reid and Heaps[35] have reported word counts on the 57,800 titles included on the MARC tapes between March 1969 and May 1970 and have noted that

(16) $\overline{W} = 5.5$

(17) $\log_{10}N = 0.6 \quad \log_{10}W + 1.2.$

Examination of other data bases has led to the conclusion that log N is likely to be a linear function of log W over the range $0 \leq W \leq 10^6$.

For a library of one million titles the Equations 16 and 17 may therefore be used to predict that when $T = 10^6$ then

(18) $W = 5.5 \times 10^6$ and $N = 1.8 \times 10^5$.

It follows from Equation 6 that if $\alpha = 1$ the number of bits required in the major is

(19) $r = 18.$

According to Equation 7, in order to reduce the frequency f of collisions at virtual addresses to 2^{-16} the number of bits required in the entire virtual address is

(20) $v = \ulcorner [\log_2 (1.8 \times 10^5 + 16 - 1] = 33.$

Consequently, the number of bits in the minor is

(21) $m = v - r = 15.$

However, with such a choice of r then $R = 2^{18}$ and the value of the load factor is, in fact,

(22) $\alpha = N/R \approx 0.7$

It follows from Equation 4 that the expected total number $\overline{p}$ of collisions at virtual addresses is equal to approximately 2. It may be further noted that Murray[36] has derived the following approximation for the probability that the number of collisions at virtual hash addresses lies within the range a to d:

(23) $$P(a, d) = \sum_{i=a}^{d} e^{-\overline{p}}\, \overline{p}^i/i! \qquad (0 \leq i \leq \llcorner \tfrac{1}{2} N)$$

where $\llcorner$ means "greatest integer less than or equal to."

When $\overline{p} = 2$ the equation gives a value of 0.9998 for the probability that the total number of collisions lies between 0 and 8. Thus the above choice of r, v, and m leads to a title word hash table file with excellent virtual address collision properties.

Use of Equation 10 with $b = 64$ and $\alpha = 0.7$, leads to the result that the probability of bucket overflow may be reduced to 0.01 by choosing $c = 62$.

In view of the above value of m it proves convenient to allocate 10 bytes of storage for each content entry. Each entry consists of a 2-byte portion to contain the 15-bit minor preceded by a collision bit, a 1-byte portion to contain a 7-bit chain pointer preceded by an overflow bit, a 3-byte dictionary pointer, and a 4-byte pointer to an inverted index. The 64 one-byte index entries, the 62 ten-byte content entries, and 4 one-byte counters, constitute buckets of length 688 bytes. The entire hash file consists of R entries, and hence $R/b = 2^{12}$ buckets. Its storage requirement is therefore for $2^{12} \times 688 = 2.82 \times 10^6$ bytes.

It may be remarked that nine 688-byte buckets may be stored unblocked in one track of an IBM 2316 disk pack, and that the entire hash file occupies 11.38 percent of the disk pack. When the disk and channel are idle the average time to access such a bucket is the sum of the average seek time, the average rotational delay, and the record transmission time. For storage on an IBM 2314 disk drive the average bucket access time is therefore $60 + 12.5 + 2.8 = 75.3$ milliseconds. The average access time for a sequence of accesses could be reduced by suitable scheduling.

SIZE OF HASH FILE FOR LC CALL NUMBERS

For a library of one million titles the number N of call numbers is 10^6. If $\alpha = 1$ and $f = 2^{-16}$ it follows from Equations 6, 7, 9, and 4 that

$$r = 20, \quad v = 35, \quad m = 15, \quad \bar{p} = 16. \tag{24}$$

With such a choice of r the load factor is approximately equal to 1. Equation 23 gives a probability of 0.9998 that the total number of virtual address collisions lies between 0 and 34. Use of Equation 10 with $b = 64$ and $\alpha = 1.0$ shows that the probability of bucket overflow may be reduced to 0.01 by choosing $c = 85$.

The content entries for LC call numbers may be arranged as for title words except that the 4-byte pointer to an inverted index is replaced by a 3-byte pointer to the compressed catalog file. The bucket length is therefore $64 + 85 \times 9 + 4 = 833$ bytes.

The storage requirement for the hash file is $(2^{20}/2^6) \times 833 = 13.65 \times 10^6$ bytes which may be stored in 2184 tracks, or 54.6 percent, of an IBM 2316 disk pack. The average time to access a bucket is $60 + 12.5 + 3.3 = 75.8$ milliseconds.

SIZE OF HASH FILE FOR AUTHOR NAMES

In the present section the term "author" will be used to include personal names, corporate names, editors, compilers, composers, translators, and so forth. It will be assumed that for personal names only surnames are entered into the author dictionary. A search query that includes specification of authors with initials is first processed as if initials were omitted, and the resulting retrieved catalog entries are then scanned sequentially to eliminate any entries whose authors do not have the required initials. It will also be supposed that each word of a corporate name is entered separately into the author dictionary, and that the inverted index contains an entry for each term.

In the absence of reliable statistics regarding the distributions of author surnames, words within corporate names, and so forth, the following assumptions have been made in order to estimate the size of the author dictionary and hash file for a library of one million titles:

(i) Personal author names contain 2×10^5 different surnames of average length 7 characters.

(ii) The corporate author names include 4×10^4 different words of average length 6 characters.

(iii) The author names include 1.6×10^4 different acronyms such as IBM, ASLIB, and so forth; their average length is 4 characters.

It is thus supposed that $N = 2.56 \times 10^5$ entries are required in the author hash files.

Calculations similar to those of the previous section show that

$$r = 18, \quad v = 33, \quad m = 15, \quad \bar{p} = 4, \quad \alpha = 1.0. \tag{25}$$

Equation 23 gives a probability of 0.9999 that the total number of virtual address collisions lies between 0 and 13. The probability of bucket overflow may be reduced to 0.01 by choosing $c = 85$. Content entries of 10 bytes may be arranged as previously described for title words. Hence each bucket requires 918 bytes of storage.

The storage requirement for the hash file is $(2^{18}/2^6) \times 918 = 3.76 \times 10^6$ bytes which may be stored in 586 tracks, or 14.6 percent, of an IBM 2316 disk pack. The average time to access a bucket is 76.1 milliseconds.

STRUCTURE OF DICTIONARY FILES

The structure of the dictionary files for title words and author names is as described by Thiel and Heaps.[37, 38] Each dictionary file contains up to 128 directories each of which points to up to 128 term strings that may each contain space for storage of 128 terms of equal length. Thus each dictionary file contains up to 2^{14} different terms. The dictionary pointers in the hash files are essentially the codes stored instead of alphanumeric terms in the catalog file.

The most frequent 127 title words are assigned dictionary pointers of the form

$$10000000 \quad 10000000 \quad \underbrace{1XXXXXXX}_{P_T} \tag{26}$$

and do not have corresponding entries in the inverted index file. The last byte forms the code used to represent the title word within the compressed catalog file.

The next most frequent 16,384 title words are assigned dictionary pointers of the form

$$00000000 \quad 1XXXXXXX \quad 1XXXXXXX \tag{27}$$

or

$$10000000 \quad \underbrace{0XXXXXXX}_{P_S} \quad \underbrace{1XXXXXXX}_{P_T} \tag{28}$$

according as there is, or is not, a corresponding entry in the inverted index. The last 2 bytes are used as codes in the compressed catalog file.

The remaining title words are assigned dictionary pointers of the form

$$(29) \qquad \underbrace{0XXXXXXX}_{P_D} \quad \underbrace{0XXXXXXX}_{P_S} \quad \underbrace{1XXXXXXX}_{P_T}$$

They all have corresponding entries in the inverted index file, and the 3 bytes are used as codes in the catalog file.

The reason that terms coded in the form 26 or 28 do not have corresponding entries in the inverted index file is that very frequently occurring terms form very inefficient search keys. Also, previous results suggest that omission of corresponding entries in the inverted index allows its size to be reduced by about 50 percent.[39, 40]

The codes of type P_T, (P_S,P_T), and (P_D,P_S,P_T) are used respectively for approximately 50 percent, 45 percent, and 5 percent of the title words. The average length of the coded title words in the compressed catalog file is therefore 1.55 bytes.

Associated with each dictionary file there is a directory of length 512 bytes whose entries point to the beginnings of term strings within the dictionary file and also indicate the lengths of the terms. Within the hash table file a dictionary pointer of the form P_D, P_S, P_T points to the P_T th term of the P_S th term string in the dictionary associated with the P_D th directory. There is a single directory associated with each set of pointers of type P_T and P_S, P_T.

The average length of the 1.8×10^5 different title words is 7.6 characters, and hence the entire set of term strings requires $1.8 \times 10^5 \times 7.6 = 1.37 \times 10^6$ bytes for storage of title words. Since twelve directories occupying $12 \times 512 = 6144$ bytes will be required, and since some term strings will contain unfilled portions, the storage requirement of the dictionary file will be slightly larger. If the title word dictionary is stored on disk in 1,000 byte records then the storage requirement is 238 tracks, or 5.95 percent, of an IBM 2316 disk pack.

The assumptions made previously regarding author names imply an author dictionary size of 1.70×10^6 bytes and sixteen directories whose total storage requirements are $16 \times 512 = 8{,}192$ bytes. Using an IBM 2316 disk pack the storage requirement is for 286 tracks, or 7.15 percent.

On completion of a search through use of the inverted index file there results a set of sequence numbers that indicate the position of the relevant items in the compressed catalog file. Before such items are displayed to a user of the system, each term must be decoded through access to the directory and dictionary to which it points.

The time required to decode a catalog item depends on how the directories and dictionaries are partitioned between disk and core memory. Several partitioning schemes for title words have been analysed, and the results are summarized in Table 2.

In the calculations used to obtain Table 2 it is assumed that title words occur with the frequencies listed by Kucera and Francis.[41] It is supposed that both the directory and term strings corresponding to codes of form P_T are stored in a single physical record, that every other directory is contained wholly within a physical record, and that each dictionary term may be located by a single access to a term string. Any required CPU time is regarded as insignificant compared to the time needed for file accesses.

From the results shown in Table 2 it appears that the best partition between core and disk is probably that which gives an average decode time of 42 milliseconds while requiring a dedicated 1501 bytes of core memory. This results when core is used to store both the directories and term strings for terms that correspond to pointers of type P_T, and the directories only for terms that correspond to pointers of type P_S,P_T.

COMPRESSED CATALOG FILE

Since the title word codes stored in the compressed catalog file have an average length of 1.55 bytes, whereas uncoded title words and their delimiting spaces have an average length of 6.5 characters, the compressed title fields occupy only 24 percent of the storage required for uncompressed words. Uncoded author names and their delimiting spaces have an average length of 7.6 characters and are coded to occupy not more than 3 bytes; hence coding of author names effects an average compression factor of less than $3/7.6 = 40$ percent. For LC call numbers the compression factor is less than 30 percent. Clearly, subject headings, publisher names, and series statements may be coded with even more effective compression factors.

The saving in space through compression of the catalog file may be translated into a cost saving as follows. If there are an average of 5.5 words in each title then one million titles include 5.5×10^6 title words and delimiting spaces which, if stored in the catalog file in uncoded form, would require 3.63×10^7 bytes.[42] When stored in coded form the requirement is for 8.54×10^6 bytes. Charges for disk space vary considerably with different computing facilities. At the University of Alberta users of the IBM 360 Model 67 are charged a monthly rate of \$.50 for each 4,096 bytes of disk storage. Thus, for title words alone the advantage of storing the catalog file in compressed form is to allow the monthly storage cost to be reduced from \$4,440 to \$950.

CONCLUDING REMARKS

The results reported in the present paper indicate that a satisfactory structure for a catalog file may be designed to use the concept of virtual hash addressing and storage of terms in compressed form. Access and decoding times may be reduced to acceptable amounts.

It may prove advantageous to arrange the items in the catalog file in the order of their call numbers. This will tend to reduce the number of disk

Table 2. Average Time to Decode a Title Word of the Compressed Catalog File.

Core Resident		*Average*	*Average*	*Dedicated*
Directories	*Term String*	*Number Accesses*	*Decode Time (milliseconds)*	*Core Memory (bytes)*
None	None	1.50	115	0
P_T	P_T	1.01	77	989
P_T, (P_S, P_T)	P_T	0.55	42	1501
All	P_T	0.50	39	7133
P_T, (P_S, P_T)	P_T, $(P_S, P_T)^*$	0.49	38	2474
All	P_T, $(P_S, P_T)^*$	0.44	34	8106

$(P_S, P_T)^*$ signifies the 128 most frequent of the codes P_S, P_T

accesses needed to retrieve catalog items in response to queries since it will tend to group relevant items. However, the benefits should be weighed against the additional expense required to maintain and update the ordered file.

The present paper has omitted discussion of the form of the query language or the search algorithm that operates on the elements of the inverted index. A formal definition of one form of query language has been discussed by Dimsdale.[43]

Details of a search algorithm and structure of a compressed form of inverted index have been discussed by Thiel and Heaps.[44] It may be noted that each content entry in the hash table file has 4 bytes reserved for a pointer to a bit string of the inverted index. Whenever the bit string is less than 4 bytes in length it is stored in the content section and no pointer is required. Storage of such bit strings within the content entries significantly reduces the storage requirements of the inverted index and also reduces the number of required disk accesses in the search phase of the program.

ACKNOWLEDGMENT

The authors wish to express their appreciation to the National Research Council of Canada for their support of the present investigation.

REFERENCES

1. D. Lefkovitz and R. V. Powers, "A List-Structured Chemical Information Retrieval System," in G. Schecter, ed., *Information Retrieval* (Washington, D.C.: Thompson Book Co., 1967), p.109–29.
2. P. R. Weinberg, "A Time Sharing Chemical Information Retrieval System" (Doctoral Thesis, Univ. of Pennsylvania, 1969).
3. R. M. Curtice, "Experimental Retrieval Systems Studies. Report No. 1. Magnetic Tape and Disc File Organization for Retrieval" (Master's Thesis, Lehigh Univ., 1966).
4. D. Lefkovitz, *File Structures for On-Line Systems* (New York: Spartan Books, 1969).

5. I. B. Holbrook, "A Threaded-file Retrieval System," *Journal of the American Society for Information Science* 21: 40–48 (Jan.-Feb. 1970).
6. G. G. Dodd, "Elements of Data Management Systems," *Computer Surveys* 1:117–33 (June 1969).
7. J. W. Rettenmayer, "File Ordering and Retrieval Cost," *Information Storage and Retrieval* 8:79–93 (April 1972).
8. R. T. Divett, "Design of a File Structure for a Total System Computer Program for Medical Libraries and Programming of the Book Citation Module" (Doctoral Thesis, Univ. of Utah, 1968).
9. H. P. Burnaugh, "The BOLD (Bibliographic On-Line Display) System," in G. Schecter, ed., *Information Retrieval* (Washington, D.C.: Thompson Book Co., 1967), p.53–66.
10. Lefkovitz, Powers, "A List-Structured Chemical Information," p.109–29.
11. Lefkovitz, *File Structures for On-line Systems*, p.141.
12. Ibid., p.177.
13. F. G. Kilgour, "Concept of an On-Line Computerized Catalog," *Journal of Library Automation* 3:1–11 (March 1970).
14. J. L. Cunningham, W. D. Schieber, and R. M. Shoffner, *A Study of the Organization and Search of Bibliographic Holdings Records in On-Line Computer Systems: Phase I* (Berkeley: Univ. of California, 1969).
15. R. S. Marcus, P. Kugel, and R. L. Kusik, "An Experimental Computer Stored, Augmented Catalog of Professional Literature," in *Proceedings of the 1969 Spring Joint Computer Conference* (Montvale: AFIPS Press, 1969) p.461–73.
16. J. W. Henderson and J. A. Rosenthal, eds., *Library Catalogs: Their Preservation and Maintenance by Photographic and Automated Techniques;* M.I.T. Report 14 (Cambridge, Mass.: M.I.T. Press, 1968).
17. I. A. Warheit, "File Organization of Library Records," *Journal of Library Automation* 2:20–30 (March 1969).
18. R. Morris, "Scatter Storage Techniques," *Communications of the ACM* 11:38–44 (Jan. 1968).
19. D. M. Murray, "A Scatter Storage Scheme for Dictionary Lookups," *Journal of Library Automation* 3:173–201 (Sept. 1970).
20. W. Buchholz, "File Organization and Addressing," *IBM Systems Journal* 2:86–111 (June 1963).
21. P. L. Long, K. B. L. Rastogi, J. E. Rush, and J. A. Wyckoff, "Large On-Line Files of Bibliographic Data: An Efficient Design and a Mathematical Predictor of Retrieval Behavior," in *Information Processing 71* (North Holland Publishing Company, 1972) p.473–78.
22. Buchholz, "File Organization," p.102–3.
23. W. P. Heising, "Note on Random Addressing Techniques," *IBM Systems Journal* 2:112–16 (June 1963).
24. Murray, "A Scatter Storage Scheme," p.178.
25. Ibid., p.181.
26. G. Schay and W. G. Spruth, "Analysis of a File Addressing Method," *Communications of the ACM* 5:459–62 (August 1962).
27. M. Tainter, "Addressing for Random-Access Storage with Multiple Bucket Capacities," *Journal of the ACM* 10:307–15 (July 1963).
28. Heising, "Note on Random Addressing," p.112–16.
29. W. H. Beyer, *Handbook of Tables for Probability and Statistics* (Cleveland: The Chemical Rubber Company, 1966).
30. Buchholz, "File Organization," p.99.
31. Morris, "Scatter Storage," p.42.
32. W. D. Maurer, "An Improved Hash Code for Scatter Storage," *Communications of the ACM* 11:35–38 (Jan. 1968).

33. J. R. Bell, "The Quadratic Quotient Method: A Hash Code Eliminating Secondary Clustering," *Communications of the ACM* 13:107–9 (Feb. 1970).
34. Morris, "Scatter Storage," p.40.
35. W. D. Reid and H. S. Heaps, "Compression of Data for Library Automation," in Canadian Association of College and University Libraries: *Automation in Libraries 1971* (Ottawa: Canadian Library Association, 1971), p.2.1–2.21.
36. Murray, "A Scatter Storage Scheme," p.183.
37. L. H. Thiel and H. S. Heaps, "Program Design for Retrospective Searches on Large Data Bases," *Information Storage and Retrieval* 8:1–20 (Jan. 1972).
38. H. S. Heaps, "Storage Analysis of a Compression Coding for Document Data Bases," *INFOR* 10:47–61 (Feb. 1972).
39. Thiel and Heaps, "Program Design," p.15–16.
40. Reid and Heaps, "Compression of Data," p.2.1–2.21.
41. H. Kučera and W. N. Francis, *Computational Analysis of Present-Day American English* (Providence: Brown University Press, 1967).
42. Reid and Heaps, "Compression of Data," p.2.4.
43. J. J. Dimsdale, "Application of On-Line Computer Systems to Library Automation" (Master's Thesis, Univ. of Alberta, 1971), p.50–68.
44. Thiel and Heaps, "Program Design," p.1–20.