Information Retrieval and Language Processing
C.A. Montgomery Editor

Attribute Based File Organization in a Paged Memory Environment

James B. Rothnie Jr. and Tomas Lozano
Massachusetts Institute of Technology

The high cost of page accessing implies a need for for more careful data organization in a paged memory than is typical of most inverted file and similar approaches to multi-key retrieval. This article analyses that cost and proposes a method called multiple key hashing which attempts to minimize it. Since this approach is not always preferable to inversion, a combined method is described. The exact specifications of this combination for a file with given data and traffic characteristics is formulated as a mathematical program. The proposed heuristic solution to this program can often improve on a simple inversion technique by a factor of 2 or 3.

Key Words and Phrases: file organization, paging, retrieval algorithm, inverted file, multiple key hashing

CR Categories: 3.70, 3.73, 3.74, 3.79

1. Introduction

A large class of information retrieval problems in attribute based files involves the accession of all records in a file which have the values of a specific set of attributes in common. Traditionally, such problems have been handled by inverted file structures, multilist file structures or some mixture of the two. The various possibilities have been described by Lefkovitz [1], Hsiao and Harary [2], and others. The fundamental property of these approaches is that it is possible to generate the set of addresses of all records having a particular value for a given attribute without accessing any records which do not have this property.

For the purpose of this article, the distinctions between traditional methods are unimportant and the inverted file structure will be assumed as a model for the class of techniques. Informally, the inverted file structure consists of two parts: the main file and the directories. The main file contains all of the records in the file. The set of directories contains one member for each inverted attribute in the file. The directory for attribute i contains all of the unique values which that attribute takes on in the file. Associated with each of these values is a list of addresses of the records in which attribute i has the given value.

A typical formulation of the inverted file approach makes no attempt to organize records in the main file according to expected usage. The implicit assumption is that references to main file addresses are homogeneous in cost, and hence, given the address of a record, the cost of retrieving the record is independent of the address value. The assumption is accurate for a truly random access device such as a core memory. However, few practical applications involve main files which can reside entirely in core. In fact, the more typical situation is one in which only a small fraction of the main file can be core resident at a time. The rest of the file must be located on a secondary storage device characterized by a much longer access time than core. Thus the assumption of cost homogeneity should usually be replaced by a cost function in which a sequence of local accesses is far cheaper than a sequence of scattered references.

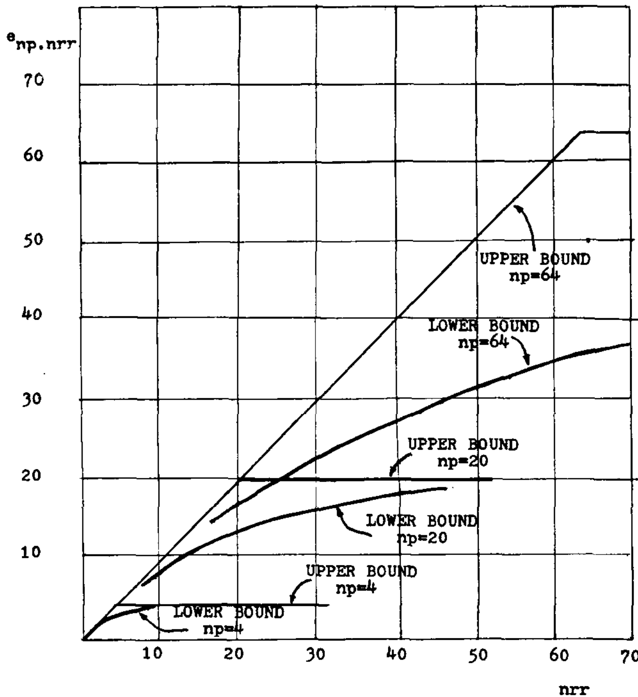
This paper will assume a paged memory environment. In such a memory, the set of addresses is partitioned into fixed size subsets called pages. If we let C_1 be the cost of an initial access of a word on a given page and C_2 be the cost of an access of a word on the same page, which occurs while the page is still in core, then $C_1 \gg C_2$. Physically, this is because data is transmitted from secondary storage to core in pages. On the initial access, the cost of retrieving the page from secondary storage must be borne, while on subsequent accesses to the same page this cost will not be incurred.

Typically, C_1 is approximately three orders of magnitude larger than C_2. In MULTICS [3], a time sharing system available at M.I.T., for example, the cpu time charge to users for each new page access is about 3 msec, while core references cost only 1–2 μsec. For this reason the number of page accesses which occur during the performance of a retrieval operation is a reasonable surrogate of the total cost. Throughout the remainder of this paper it will be assumed that the performance measure which retrieval algorithms are attempting to minimize is the total number of page accesses required to complete an opera-

Copyright © 1974, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

This work was supported in part by the Cambridge Project under Contract DAHC15 69 C 034 of the Advanced Research Projects Agency. The MIT Civil Engineering Systems Laboratory also provided support for this work. Authors' current addresses are: T. Lozano: Artificial Intelligence Laboratory, MIT, Cambridge, MA 02139; J. B. Rothnie Jr.: 7058 Darnell Dr., Fayetteville, NC 28304.

Fig. 1. Bounds on expected page accesses.



tion. The assumption is useful because it leads to reasonably tractable expressions for the performance of the techniques of interest.

This paper is concerned with structuring the main file in such a way that accesses will be minimized with respect to a probabilistic statement of anticipated retrieval requests. The discussion begins with a formalized description of the file structure. This is followed by an analytic consideration of the need for structuring the main file. Next, a structuring methodology is presented and a retrieval algorithm is described which combines the methodology with the inverted approach. Finally, the problem of specifying the various parameters in this technique is formulated as a mathematical program which minimizes page accessing for a limited class of retrieval requests. A heuristic solution of this program produces results which can often improve on a simple inversion technique by a factor of 2 or 3.

2. File Structure

The file structure used in this article will be a modification of the model suggested by Hsiao and Harary [2]. In this formulation a *file* is defined as a named set of records. The file F is associated with a set of *attributes* A_F. Each record in F is a set of ordered pairs of the form (a_{F_i}, v_j) where a_{F_i} is an element of A_F and v_j is a value. An ordered pair of this type is called a *keyword.* A keyword of the form (a_{F_i}, v_j) will be designated K_{ij}. For each element of A_F, say a_{F_i}, a *record* contains exactly one keyword in which the first element is a_{F_i}.

Each record, r, is associated with a unique integer called its *address.* In the formal system of Hsiao and Harary, addresses were simply identifiers for records and the set of addresses had no other structure. In this paper, however, the set of addresses is partitioned by the set of pages.

In this paper it will be assumed that users designate the records of interest by specifying a keyword which must be an element of any selected record. This is a special case of Hsiao and Harary's formulation in which records were designated by descriptions, that is, combinations of keywords. (The technique described in this paper can be readily generalized to this type of request, but the analysis of the technique becomes much more difficult.) The operator R maps a keyword K_{ij} into the set of records which contain that keyword, and the operator I maps K_{ij} into the set of addresses of records in $R(K_{ij})$. Finally, the operator P maps a set of addresses into the set of pages on which they occur. That is, $P(I(K_{ij}))$ is the set of pages on which records containing K_{ij} occur.

3. The Need for Structuring

The next step in this discussion is to explain the need for structuring by computing the expected number of page accesses which will occur for files of various sizes subjected to various numbers of record retrievals.

Figure 1 shows the results of several such computations. File size, *np*, is the number of pages on which records in the file are located. *nrr* is the number of record retrievals from the mainfile. Assuming that these accesses are randomly distributed over the address space, $e_{np,nrr}$ is the expected number of page accesses which will occur in retrieving *nrr* records from a file of *np* pages. The two curves shown for each file size represent upper and lower bounds on $e_{np,nrr}$. The upper bound was computed by assuming that no two record retrievals will be drawn from the same page (until all pages have been accessed). The lower bound is based on the assumption that the probability of a retrieval from a given page is not reduced by the fact that there have been previous retrievals from the page. Note that these curves are bounds on expected page accesses and not on actual page accesses. The upper bound will be approached as the number of records per page approaches 1, while the lower bound will be approached as the number of records per page approaches infinity.

Consider, for example, a file containing 64 pages, each with 100 records. Suppose that a user's retrieval request, K_{ij}, has been processed through the directory of attribute A_{F_i} and results in the need to access 20 records in the main file. If records are randomly distributed over the entire address space, then a lower

bound on expected page accesses is 17.3 and an upper bound is 20.

This is the performance which can be expected if records are randomly distributed over the entire address space. That is, there is an equal probability that a given record containing K_{ij} will occur on any page of the address space.

Now suppose that the file has been organized so that there are only four pages with a nonzero probability of containing a record satisfying K_{ij}. In that case, it is clear that the number of page accesses required to process the request is upper bounded at 4. In such circumstances, we say that the "effective" size of the file is 4 relative to the particular user's request.

The effective size of a file is defined relative to a keyword K_{ij}. The term refers to the total number of pages for which the probability of finding a record which satisfies K_{ij} is greater than zero. $S(K_{ij})$ is defined as the set of such pages and the cardinality, $\#(S(K_{ij}))$, is the effective size. In this article, the qualifying records are assumed to be randomly distributed over $S(K_{ij})$.

A new interpretation of Figure 1 will indicate the importance of effective size. Let np represent not the full size of a file but the effective size relative to a description, that is, $\#(S(K_{ij}))$. The reader should observe the strong impact of reduced effective size on retrieval cost. Again, note the difference between the curves for $np = 64$ and $np = 4$. The smaller effective size improves performance (i.e. reduces page accesses) by a factor of 4. We will now consider a methodology for organizing the main file in such a way that its effective size will be reduced relative to keywords expected to occur in user requests.

4. Reducing Effective Size—Multiple Key Hashing

The desired shrinkage of effective size can be accomplished by a variation on the look-up technique, termed hashing or scatter storage. This method will be called multiple key hashing or *mkh*. (A related scheme was proposed in a different context by Gustafson [4].)

In multiple key hashing some chosen set of attributes called the *mkh* set is selected as the basis for the main file organization. If a_{F_i} is an element of that set, it will be associated with a hashing function, h_i, which maps the values which a_F can take on into integers called *subfile indices*.

Every record in the file F can be mapped into an object called its *characteristic tuple*. A characteristic tuple is a tuple (ordered set) of the form $(h_1(v_1), h_2(v_2), \ldots, h_m(v_m))$ where the attributes a_{F_1} through a_{F_m} are elements of the *mkh* set and v_1 through v_m are the values which those attributes take on in the record of interest. A set of records every member of which maps into the same characteristic tuple is called a cluster. *The cluster is the organizational unit of multiple key hashing; members of the same cluster are stored on the same page of the address space.*[1] By selecting appropriate h_i functions, one can limit the total number of clusters on which records containing a particular keyword will be found. Since clusters are stored on a single page, this technique also limits the effective size of the file relative to that keyword.

Clearly, then, the selection of hashing functions is a key problem. For the purposes of this paper, a hashing function h_i has two important characteristics. First is the distribution of records over the subfile indices which it defines. Since pages are of a fixed size, hashing functions should be chosen in such a way that this distribution is approximately uniform. (For a discussion of what constitutes a reasonable approximation of uniformity see [5].) Since the uniformity objective is also important in single key hashing, the problem has been given a great deal of attention (for example, see [6]) and will not be further considered here.

The second significant characteristic of a hashing function, h_i, is the number of subfile indices in its range, that is, the number of subfile indices into which it maps the values of a_{F_i}. This parameter will be called $nfile_i$. From the definition of a characteristic tuple it is clear that the number of characteristic tuples is $\prod_{i=1}^{m} nfile_i$ where m is the number of attributes in the *mkh* set. Since there is a one-to-one correspondence between clusters and characteristic tuples, the number of clusters, nc, is defined by the relation:

$$nc = \prod_{i=1}^{m} nfile_i. \quad (1)$$

As we will see, the values of $nfile_i$ have a fundamental impact on the effective size of a file relative to a keyword. An important problem in *mkh* is the selection of good $nfile_i$ values. As the next paragraphs will indicate, these values are limited by a tight constraint.

The *mkh* technique partitions the set of records in a file into logical groups called clusters, such that all the elements of a given cluster have certain characteristics in common. An early premise of this paper is that physical memory is partitioned into groups of addresses called pages, and that a reasonable objective of a search algorithm is to minimize the number of pages accessed in retrieving the records containing some keyword. To attain this objective two conditions must be satisfied:

1. Records must be allocated to pages in such a way that a small number of pages will contain records satisfying the keyword.
2. There must exist some mechanism for determining which pages contain these records without accessing other pages.

If all of the members of a cluster are stored on the same page, then both these conditions can be fulfilled.

[1] The exact number of elements in each cluster is a function of the distribution of values in the file. Hence this parameter cannot be precisely controlled by the file designer, and it is not possible to guarantee that all elements of a cluster will appear on the same page. The goal in designing hashing functions is to approximate this desired situation.

For the objective function of this article the organization of records within a page is irrelevant. To simplify the analysis we have assumed that the cost of accessing one record in a page is the same as accessing all records. Hence, there is no benefit to be gained by allocating more than one cluster to a page. If we assign exactly one cluster to a page, then:

$$np = nc \tag{2}$$

and from eq. (1):

$$np = \prod_{i=1}^{m} nfile_i . \tag{3}$$

This relation would be exactly correct if the $nfile_i$'s could take on noninteger values. However, since these variables must be integers the relation should be expressed as:

$$np \cong \prod_{i=1}^{m} nfile_i . \tag{4}$$

A discussion in Section 6 will describe an algorithm for assigning *nfile* values which meet this approximate constraint. For the remainder of the paper, the relation will be expressed as an equality.

The value of *mkh* in reducing effective size can be seen by considering a keyword K_{ij} (or(a_{F_i},v_j)). The effective size of the file relative to that description is ($np/nfile_i$). This is the number of clusters in which the subfile index for attribute a_{F_i} is $h_i(v_j)$. A large $nfile_i$ value can therefore substantially reduce the page accessing which will occur relative to K_{ij} .

Fig. 2. File organized by multiple key hashing.

a_{F_1}	a_{F_2}	a_{F_3}	Characteristic Tuple	Page Number
1 9	1 1	1 2	(1,1,1)	1
1 5 5 9	1 1 1 1	3 3 4 3	(1,1,2)	2
1	2	1	(1,2,1)	3
1 5 9 9	2 2 2 2	3 3 3 4	(1,2,2)	4
1 5 9	3 3 3	2 1 1	(1,3,1)	5
9	3	2	(1,3,2)	6
14 14	1 1	1 2	(2,1,1)	7
12 20	1 1	4 3	(2,1,2)	8
12	2	2	(2,2,1)	9
12 14 14	2 2 2	3 3 4	(2,2,2)	10
12 14 20	3 3 3	1 1 2	(2,3,1)	11
14 20	3 3	3 4	(2,3,2)	12

$h_1(x) = 1$ if $1 \le x \le 10$
$= 2$ if $11 \le x \le 20$
$nfile_1 = 2$
$h_2(x) = x$
$nfile_2 = 3$
$h_3(x) = 1$ if $1 \le x \le 2$
$= 2$ if $3 \le x \le 4$
$nfile_3 = 2$

5. A Retrieval Algorithm Based on Multiple Key Hashing

In addition to reducing effective file size relative to a keyword, *mkh* can be used to completely eliminate the need for an inverted file in certain situations. This is because the main file organization gives the system a limited capability to identify those parts of the address space which may contain records with a certain desired content. Given a keyword, K_{ij} , it is possible to identify those $np/nfile_i$ pages which may contain a qualified record. This can be accomplished by: (1) generating the set of characteristic tuples which have $h_i(v_j)$ as the ith element (let *CT* be a function which maps a keyword into such a set); and (2) mapping those tuples into the corresponding pages (let *CP* be a function which performs that mapping). (The reader should note $CP(CT(K_{ij})) = S(K_{ij})$). By using these operators it is possible to identify pages which may contain qualified entries without making recourse to inverted file directories.

Consider for example the file depicted in Figure 2. This file is structured by multiple key hashing. It consists of three attributes, a_{F_1} , a_{F_2} , and a_{F_3} , associated with the hashing functions h_1 , h_2 , and h_3 , respectively. By using these hashing functions, 12 unique characteristic tuples can be generated, and hence the file contains 12 pages. The characteristic tuple associated with each page is shown in the diagram. Suppose, now, that we wish to retrieve all those records in which $a_{F_2} = 2$. The first step is to compute the smallest set of characteristic tuples which will define all clusters containing such records. The *CT* function for this particular case is defined as follows:

$$CT((a_{F_2}, 2)) = \{(x, y, z) \text{ st } x \in \{1, 2\} \ \& \ y = h_2(2) \ \& \ z \in \{1, 2\}\}. \tag{5}$$

The set of characteristic tuples which this function defines is {(1,2,1), (1,2,2), (2,2,1), (2,2,2)}. The next step is to compute the set of page numbers associated with these tuples. For the example, this is accomplished by the following *CP* function:

$$CP(CT(K_{ij})) = \{p \text{ st } (x, y, z) \in CT(K_{ij}) \ \& \ p = 1 + (x-1) * nfile_2 * nfile_3 + (y-1) * nfile_3 + (z-1)\}. \tag{6}$$

For the keyword chosen, this set of pages is {3,4,9,10}.

The interested reader can readily generalize the *CT* and *CP* functions to arbitrary keyword retrievals.

The advantages which this approach offers over inversion fall into two categories. First, *mkh* will consume a negligible amount of storage beyond that needed for the main file. Inversion, on the other hand, requires significant extra space. Second, the computation of the set of clusters to be searched can be accomplished with few page accesses since only the data associated with the hashing functions need be retrieved. The inverted file approach requires storage references to obtain the $I(K_{ij})$ sets which are required in the computation.

On the other hand, there are some situations in which a standard inverted file retrieval algorithm is preferred to the *mkh* search scheme. This occurs for descriptions involving small *nfile* values. In such cases, the quotient $np/nfile_i$ will be large and hence many page accesses will be required if *mkh* is employed. The reader should note that *nfile* values are sharply constrained by the relation $\prod_{i=1}^{m} nfile_i = np$, and hence, in nearly every file there will be some attributes whose *nfile* values are small.

Since there are some situations in which the *mkh* retrieval scheme should be used and others in which inversion is preferred, a combined approach is suggested. This hybrid algorithm will partition a file's attributes into two subsets, one to be handled by *mkh* and the other by inversion. We are assuming that the intersection of these subsets is null. (Equivalently, it can be said that all attributes are in the *mkh* subset, but all those which are also in the inverted file subset have an *nfile* value equal to 1.) This assumption will be useful in deriving a near optimal combination of the two methods in the next section.

6. An Optimal Combination of Inversion and Multiple Key Hashing

Up to this point we have described a mechanism for organizing records in the main file according to content and for using this organization, in conjunction with inversion, as the basis for a retrieval algorithm. The use of this technique, however, involves a number of detailed decisions which must be made for each file. The decisions fall into two basic categories:

—Which attributes should belong to the *mkh* set and which to the inverted set?

—What should the *nfile* values be for the attributes in the *mkh* set?

The discussion has not yet considered how these decisions are made.

In the past, decisions of this nature were made by the file designer. He used his general knowledge of the trade-offs involved in a file structure and some feeling for the future use of the file to determine qualitatively what the specific file characteristics should be. While this method might be adequate for an experienced file designer, it presents obvious difficulties to the novice. Furthermore, that approach is difficult to embody in coding for automatic file design.

In this article, a more rigorous framework for making the basic file organization decisions will be presented. Specifically, these decisions will be formulated as in the following mathematical optimization problem.

$$\text{Min} \sum_{i=1}^{na} a_i p_i (np/nfile_i + 1) + \sum_{i=1}^{na} b_i p_i (e_{np,nr/nv_i} + 2), \quad (7)$$

such that $\prod_{i=1}^{na} nfile_i = np$; $a_i + b_i = 1$; $nfile_i$, a_i, b_i are integers; and $nfile_i$, a_i, $b_i \geq 0$, where:

- nr = the number of records in the file.
- np = the number of pages in the file.
- na = the number of attributes in the file.
- nv_i = the number of unique values which a_{F_i} assumes in the file.
- p_i = the probability that a random single keyword description will be K_{ij}.
- $nfile_i$ = the number of elements in the range of h_i (i.e. h_i maps all values which a_{F_i} takes into $nfile_i$ different values).
- $e_{np,nrr}$ = the expected number of pages which will be accessed in performing *nrr* record retrievals from a file of *np* pages.
- a_i = a binary variable which is 1, if a_{F_i} is in the *mkh* set, and 0 otherwise.
- b_i = a binary variable which is 1, if a_{F_i} is in the inverted set, and 0 otherwise.

The decision variables in the formulation are a_i, b_i, and $nfile_i$. The objective function consists of two summations, each corresponding to the contribution to page accesses produced by the attributes in one of the two subsets. Within the first summation, $\sum_{i=1}^{na} a_i p_i (np/nfile_i + 1)$, the expression $(np/nfile_i + 1)$ represents the number of page accesses which will occur in processing the single keyword description K_{ij}, if a_{F_i} is in the *mkh* set and has a certain value for $nfile_i$. The 1 in that expression is *mkh* overhead. The role of a_i in the summation is to exclude any attributes not in the *mkh* set. p_i weights the contributions of attributes according to the probability of their appearance in a retrieval request. The set of p_i's constitutes a statistical description of expected user requests. These values could be initialized by a user estimate and then updated by system monitoring of actual traffic.

In the second summation, $\sum_{i=1}^{na} b_i p_i (e_{np,nr/nv_i} + 2)$, the expression $(e_{np,nr/nv_i} + 2)$ represents the page accesses which will occur for requests involving items in the inverted set. nr/nv_i is the average number of records which will have one particular value v_j in keyword K_{ij}. Again, b_i excludes attributes which are not in the inverted set, and p_i is a weighting factor.

The first constraint, $\prod_{i=1}^{na} nfile_i = np$, represents the restriction on the number of clusters. $a_i + b_i = 1$

simply indicates that an attribute cannot reside in both sets.

The form of this optimization problem makes it difficult to solve using any of the standard algorithms. A branch and bound solution was attempted, but this proved quite costly in many cases. A heuristic algorithm yielded nearly as good results for a much lower price. The latter methodology will be described here.

The heuristic can be summarized as follows:

Step 1. Choose a "reasonable" assignment of a_i and b_i values.
Step 2. Given these values, compute appropriate *nfile* values.
Step 3. Compute the value of the objective function.
Step 4. Has the heuristic stopping condition been met?
Step 5. If so, quit.
Step 6. If not, go to 1 and get a new assignment for the a_i's and b_i's.

We will begin a more detailed account of the heuristic with Step 2, computing *nfile* values given the values of a_i and b_i. This problem is another optimization problem which can be formulated as follows:

$$\text{Min} \sum_{i=1}^{na} a_i p_i \, np/nfile_i \tag{8}$$

such that

$$\prod_{i=1}^{na} nfile_i = np.$$

If we assume that the *nfile* parameters can take on continuous values, this problem can be solved by a Lagrangian method.

L is defined by the following expression:

$$L = \sum_{i=1}^{na} a_i p_i \, np/nfile_i - \lambda\left(\prod_{i=1}^{na} nfile_i - np\right). \tag{9}$$

The first order conditions for optimality are defined by the following system of equations:

$$\partial L/\partial nfile_i = a_i p_i np_2/nfile_i - \lambda \prod_{\substack{j=1 \\ j \neq i}}^{na} nfile_j = 0, \quad \text{for all } i,$$

$$\partial L/\partial \lambda = \prod_{i=1}^{na} nfile_i - np = 0. \tag{10}$$

The solution of the system leads to the following optimality conditions:

$$p_i/nfile_i = p_j/nfile_j \tag{11}$$

for all i and j, such that a_i , $a_j = 1$ (that is, for all attributes in the *mkh* set).

An algorithm which produces an integer approximation of optimality is the following:

1. Set $nfile_i = 1$, for all i.
2. Choose j such that $a_j p_j/nfile_j \geq a_i p_i/nfile_i$, for all i.
3. $nfile_j = nfile_j + 1$.
4. If $\prod_{i=1}^{na} nfile_i \geq np$, then quit; else go to 2.

(This algorithm assumes that, for all i, $nfile_i \leq nv_i$. If this condition does not hold for some attributes a_{F_j} , then $nfile_j$ should be set to nv_j.)

This algorithm will produce reasonable *nfile* values given a partition of the attributes into *mkh* and inverted sets. The remaining task is to describe a mechanism for determining the a_i and b_i values which define the attribute partition. The heuristic method used to determine these values is based on guessing a set of a_i , b_i values and then examining the resulting value of the objective function. If this value reflects an improvement over previous assignments, then an additional guess will be made in an effort to further improve the solution. As soon as expected performance begins to deteriorate, however, searching is discontinued and the best solution tried up to that point will be chosen.

The process of guessing a good potential solution is based on the principle: those attributes with high p_i values are most likely to perform well in the *mkh* set.[2] Those attributes with high nv_i values are most likely to do well in the inverted set.[3]

In generating the sequence of guesses, this rule will be used to guide the choice of trial solutions.

Specifically, the heuristic solution is as follows:

1. Begin with all attributes in the inverted set, except for those for which $e_{np,nr/nv_i} = np$. For this latter group, inversion will never yield the paging reduction to justify its overhead. Compute the expected number of page accesses for this solution and set the upper bound to this value.
2. Next order the attributes in the inverted set in descending order according to the value of the expression $p_i(e_{np,nr/nv_i} + 2)$. Call the first attribute att_1 , the next att_2 , etc. Let $i = 1$.
3. Place att_i in the *mkh* subset.
4. Compute the expected paging for this solution.
5. If this value is less than upper bound, set upper bound to the value; set $i = i + 1$; go to 3. If this value is greater than upper bound, return att_i to the inverted subset and quit.

This algorithm can be simply described as: (a) first ordering attributes according to their $p_i(e_{np,nr/nv_i} + 2)$ values; and (b) then including them, one after another, in the *mkh* subset as long as the solution is being improved.

The expression $p_i(e_{np,nr/nv_i} + 2)$ represents the contribution to paging of an attribute included in the inverted set and, therefore, constitutes an upper bound on the improvement which can occur if this item is moved to the other set. The heuristic uses this value as an indicator of *potential* improvement and tries those attributes with the greatest possibilities first. If a higher potential attribute fails to produce an improvement, no

[2] Recall that an optimal assignment of *nfile* values has the ratio $p_i/nfile_i$ equal for all attributes in the *mkh* set. Thus high p_i values correspond to high nfile values and a correspondingly low value for $np/nfile_i$.

[3] Recall that the upper bound on page accesses for a single keyword description involving an inverted attribute is nr/nv_i.

Table I. Performance of the Heuristic Solution to the Inversion/ Multiple Key Hashing Partitioning Problem

Example number	p_1	nv_1	p_2	nv_2	p_3	nv_3	p_4	nv_4	Expected paging for: Inverted file	Heuristic comb.	Optimal comb.
1	.5	64	.3	200	.13	300	.07	200	46.0	11.7	11.7
2	.5	750	.3	400	.1	1300	.1	6400	11.3	6.8	6.8
3	.3	324	.1	108	.3	640	.3	800	16.1	9.5	9.5
4	.4	1200	.3	900	.2	600	.1	300	10.0	7.9	7.9
5	.445	64	.445	430	.11	108	.0	1	40.5	12.1	10.9

further attempts are made. Clearly, the magnitude of potential improvement is not the same as the actual gain, and for that matter, the ordering of attributes by potential is not the same as an ordering by actual improvement. This is the reason that the heuristic will not necessarily yield the optimal solution. Empirical results illustrated in the examples below indicate that in most cases the difference between the paging performance of this solution and that of the optimal will not be sufficiently different to justify further search.

The results of the application of this heuristic to five example files are shown in Table I. In each case, the file contained 64 pages and 6400 records. For each set of file parameters, several alternative partitions were tested. In each case, the value of the objective function was computed by approximating $e_{np,nrr}$ by its lower bound. This will slightly bias the solutions in favor of the inverted file.

These tables illustrate two important points:

1. First of all, the heuristically chosen combinations of techniques often perform substantially better than the more traditional approach, inversion. This is an important result because it suggests that the efficiency of methods used in standard practice can be substantially improved. Note that in several cases the expected page accessing of this partition exceeds the heuristically chosen solution by a factor of 2 or 3.
2. In four of the five examples, the optimal solution was selected by the heuristic. The one exception, example five, exhibits an optimal performance of 10.9 expected page accesses, while the heuristically chosen solution yields 12.1 accesses. This rather small differential can be shown to be approaching a bound on the suboptimality of an heuristic solution to three attribute files of this size.

7. Summary

This article has addressed a number of issues:

1. First, the importance of the structuring of records in the main file was discussed. It was demonstrated that, in an unstructured file, page accesses will increase in a near linear way with record retrievals until the number of retrievals approaches the number of pages in the file. This led to the conclusions that: (a) inversion and other retrieval reducing schemes are not very useful if the number of accessed records cannot be reduced below the number of pages in the file; and (*b*) page accesses could be reduced if structuring of the file could reduce its effective size relative to a retrieval request.
2. A technique which extended hashing concepts to the randomizing of several attributes was suggested as a method of structuring the file. The approach, called multiple key hashing, organizes records into clusters which correspond roughly to pages.
3. A retrieval algorithm based on multiple key hashing, *mkh*, was described. It was observed that neither this algorithm nor the inverted file approach can produce optimal performance for all files. Hence a technique which combines the two approaches seems appropriate.
4. A combination by which certain attributes are handled by *mkh* and the others are handled by inversion was suggested. The best way to partition the attributes into these two sets was formulated as a mathematical program and a heuristic approximation to the solution was proposed. The technique yields near optimal combinations of *mkh* and inversion for a relatively small cost. When applied to several example files, this solution substantially improved on the page accessing performance of inversion used independently.

An important issue which has not been explored in the work reported here is the degradation of retrieval performance as file parameters change over a period of time. It is clear that variations in the nature of user requests and changes in attribute value distributions will impact on performance. However, the definition of a formal mechanism for establishing the values of decision variables enables the system designer to incorporate an automatic reorganization feature into the system. In this way the algorithm is able to adjust to changes in retrieval and maintenance traffic and reduce the damaging impact. The nature of the technique's degradation is nevertheless a significant area for future research, because this is the issue which determines reorganization strategy.

Received April 1973; revised September 1973

References

1. Lefkovitz, D. *File Structures for On-Line Systems*. Spartan Press, Washington, D.C., 1969, pp. 126–129.
2. Hsiao, D., and Harary, F. A formal system for information retrieval from files. *Comm. ACM 13*, 2 (Feb. 1970), 67–73.
3. Corbato, F.J., and Vyssotsky, V.A. Introduction and overview of the MULTIC system. Proc. AFIPS 1965 FJCC, Vol. 27, AFIPS Press, Montvale, N.J., pp. 185–196.
4. Gustafson, R.A. Elements of the randomized combinatorial file structure. Proc. Symp. on Information Storage and Retrieval, ACM, New York, Apr. 1971, pp. 163–174.
5. Rothnie, J.B. The design of generalized data management systems. Unpublished Ph.D. Diss., Dep. of Civil Eng., M.I.T., 1972, pp. 72–89.
6. Knuth, D., *Sorting and Searching, The Art of Computer Programming*. Vol. 3, Addison-Wesley, Reading, Mass., 1973, pp. 506–542.