Programming and Data Structures

M.D. McIlroy Editor

Partial-Match Retrieval Using Indexed Descriptor Files

John L. Pfaltz and William J. Berman
University of Virginia

Edgar M. Cagley
General Services Administration

In this paper we describe a practical method of partial-match retrieval in very large data files. A binary code word, called a descriptor, is associated with each record of the file. These record descriptors are then used to form a derived descriptor for a block of several records, which will serve as an index for the block as a whole; hence, the name "indexed descriptor files."

First the structure of these files is described and a simple, efficient retrieval algorithm is presented. Then its expected behavior, in terms of storage accesses, is analyzed in detail. Two different file creation procedures are sketched, and a number of ways in which the file organization can be "tuned" to a particular application are suggested.

Key Words and Phrases: bit codes, storage access cost, multiattribute, partial-match, retrieval

CR Categories: 3.74, 4.33, 4.34

W.J. Berman's research was supported in part by NASA contract NAS1-14862, J.L. Pfaltz's by Battelle contract DAAG29-76-D-0100.

Authors' present addresses: J.L. Pfaltz and W.J. Berman, Dept. of Applied Mathematics and Computer Science, University of Virginia, Charlottesville, VA 22903; E.G. Cagley, Mathematics and Computation Laboratory, General Services Administration, Charlottesville, VA 22903.

1. Introduction

In this paper we are concerned with partial-match retrieval [10] over large, on-line data files. Partial-match retrieval (sometimes called "retrieval by secondary keys") assumes that a set of attributes has been associated with the records of a file. A specific record will have a distinct (but not necessarily unique) value associated with each attribute. A partial-match query specifies values for one or more attributes, thus defining a subset of the file whose records have the specified values for designated attributes.

In [13] several techniques for partial-match retrieval are reviewed and four particular file structures are described: record list (also known as "multilist"), cellular list, record inverted list (often called "inverted files"), and cellular inverted list. It is stated that "the inverted list structure is generally preferred" [13, p. 266]. This preference for the inverted file structure is reflected in its use in such systems as ADABAS [6], System 2000 [6], and SPIRES[12].

Because the technique of inverted files is so widely used and understood, it is useful as a benchmark by which to measure alternative techniques. One well-known aspect of inverted files is that "in large, highly inverted databases, the inverted directory, or index, becomes a large database itself" [5, p. 262]. In fact, "exhaustive indexes for all attributes will easily exceed the size of the original file" [15, p. 133]. Therefore, only those attributes which are deemed important enough are inverted. The resulting storage overhead has been observed to range from 10 percent to 70 percent of the original file size (interpreting results from [5]). Of course, with fewer attributes being inverted, the average retrieval efficiency decreases and "very few authors have suggested practical and specific guide lines for selecting the inversion keys optimally" [5, p. 261].

Another important, but possibly not as well-known, characteristic of inverted files is that "the amount of work [for retrieval] *grows* with the number of keys given, while the expected number of records satisfying the query *decreases*!" [10, p. 25]. This observation is confirmed by the results presented [5].

As database size and query complexity have increased, many researchers have sought alternatives to the inverted file technique. In [14], Vallarino develops a framework for describing and comparing alternative techniques that use bit-strings as compact representations of data records. One alternative which Lefkovitz explores is that of using superimposed coding [7, 8]. He concludes that "the superimposed code, employed in sequential, highly compacted search files bears further investigation even though, for interactive search of large files, it is not yet as efficient as the [inverted file technique]" [9, p. 10]. Roberts [11] has pursued the practical use of superimposed code descriptors developing "bit slice," as opposed to "bit string," search algorithms (see also [14]) which may be either hardware or software implemented.

In this paper, we present yet another approach to the use of bit strings for partial-match retrieval. This approach, which we call "indexed descriptor files," fits nicely within Vallarino's framework. However, this method was originally developed in 1969 by E. Cagley, and Cagley and his associates (D. Parrish, N. Ray, and R. Vaughan) have spent over seven years using, refining, and testing it.

In the next section, the technique of "disjoint coding" is defined and its simplest use is described. Section 3 extends the simple concepts of disjoint coding to derive the file structure and retrieval algorithm for indexed descriptor files. Having described the retrieval algorithm, its expected behavior is analyzed in Section 4. This analysis includes exact and approximate expressions for the number of expected storage accesses, together with observed results from a real application of the technique. In Section 5, the many possible algorithms for creating and maintaining indexed descriptor files are divided into "top-down" and "bottom-up" classes. Finally, Section 6 summarizes the results and remaining open questions regarding indexed descriptor files.

2. Descriptors Created by Disjoint Coding

A *descriptor D* is simply a bit-string of w (for width) bits. Each record, R, in a data file has associated with it a descriptor, D_R. This descriptor is derived from the values $(v_1, v_2, \ldots, v_f)_R$ of the f attributes of R. While the particular method of creating a descriptor to reflect the attribute values of a record is not critical in the discussion of retrieval using indexed descriptor files, it may significantly affect the efficiency of an implementation. A traditional approach for creating descriptors is that of superimposed coding [7, 8, 11]. We consider another possibility, disjoint coding.

Disjoint coding begins by dividing each descriptor into f disjoint *fields* (note that each record has f attributes). Each field F_j consists of w_j bits and, therefore,

$$\Sigma_{j=1}^{f} w_j = w \tag{2.1}$$

Each of the f attributes has associated with it a transformation, T_j: {legal values of attribute-j} $\rightarrow [1, w_j]$. Then, to encode or describe a record R, these transformations are applied to each of the attribute values of R and the $T_j(v_j)^{\text{th}}$ bit in F_j is set to 1 (the remaining $w_j - 1$ bits are cleared to 0) for $j = 1, 2, \ldots, f$. For an example of disjoint coding, see Figure 1.

As just described, each transformation T_j is single-valued and, therefore, each descriptor will have exactly f bits set to 1 (one in each of its fields). Equally possible are transformations which set 2, or more, bits per field. One may also use order-preserving transformations that, with a modification to the retrieval algorithm presented in this paper, allow for "range searching."

Before developing a partial-match retrieval algorithm, it is useful to consider separately the creation of the query descriptor, Q, and its basic properties. In a partial-match query, attribute values are specified for only a subset of the attributes. Let $q \leq f$ attribute values be specified. The query descriptor Q is created in a manner analogous to that of creating D_R from R. That is, the transformation T_j is applied to the attribute value v_j' to determine which bit in $F_j(Q)$ to set to 1. If for some attribute j, v_j' is not specified in the query then *no* bit is set in $F_j(Q)$. Consequently, precisely q bits will be set in the descriptor Q.

Since the descriptors D_R and Q have been constructed using the same transformations, it is apparent that the following proposition and obvious corollaries are true.

PROPOSITION 2.0. *If R satisfies the partial-match query, then* $Q \subseteq D_R$.

COROLLARY 2.2. *If* $Q \not\subseteq D_R$, *then R does not satisfy the partial-match query.*

COROLLARY 2.3. *If* $Q \subseteq D_R$, *then R may, or may not, satisfy the partial-match query.*

Here the notation $Q \subseteq D_R$ means that every bit position which is 1 in Q is also a 1 in D_R, and the notation $Q \not\subseteq D_R$ means that there is at least one bit position which is 1 in Q and 0 in D_R. These propositions form the basis for all our retrieval algorithms.

Knowing how to create query descriptors, we may illustrate these basic properties by first constructing a trivial file structure that supports straightforward partial-match retrieval. The data records are stored in a data structure allowing random access to any record R. The descriptors are stored sequentially. The partial-match retrieval begins by forming the query descriptor Q. The descriptor file is then sequentially scanned and each descriptor D_R is bit-wise compared with Q. If $Q \subseteq D_R$, then the record R is accessed and its actual attribute

Fig. 1. Example of disjoint coding.

$$R_k = \left\{\begin{array}{ll} v_1 \text{ (NAME)} & = \text{BERMAN, WILLIAM JOSEPH} \\ v_2 \text{ (BIRTHDATE)} & = 48/8/17 \\ v_3 \text{ (EMPLOYEE \#)} & = 326 \\ v_4 \text{ (DEPARTMENT \#)} & = 34 \end{array}\right\}$$

$$\begin{array}{rl} w_1 = & 5 \text{ bits} \\ w_2 = & 3 \text{ ``} \\ w_3 = & 9 \text{ ``} \\ w_4 = & 7 \text{ ``} \\ \hline w = & 24 \text{ bits} \end{array}$$

$$T_1 = \begin{cases} 1, \text{ if first letter is A–C} \\ 2, \text{ if `` `` `` D–J} \\ 3, \text{ if `` `` `` K–N} \\ 4, \text{ if `` `` `` O–T} \\ 5, \text{ if `` `` `` U–Z} \end{cases}$$

$$T_2 = \begin{cases} 1, \text{ if before 1930} \\ 2, \text{ if between 1930 and 1950} \\ 3, \text{ if after 1950} \end{cases}$$

$$T_3 = \text{EMPLOYEE \# (modulo 9)} + 1$$

$$T_4 = \text{DEPARTMENT \# (modulo 7)} + 1$$

$$D_k = \overbrace{10000}^{F_1}\ \overbrace{010}^{F_2}\ \overbrace{001000000}^{F_3}\ \overbrace{0000001}^{F_4}$$

values are compared with those specified in the query. If $Q \not\subseteq D_R$, then by Corollary 2.2, R cannot possibly satisfy the query and, therefore, need not be accessed.

Although we shall find that the use of "indexed" descriptor files is far superior (in terms of disk accesses) to this simple version of the partial-match algorithm, it can be seen that if the descriptors are very much shorter than their corresponding records, then a sequential search of a descriptor file coupled with random access into the data file might be superior to a simple sequential search of the data file itself [9]. This comparison can be especially favorable if the descriptor file and the data file are stored on separate devices, thereby significantly reducing seek time. Another consideration in favor of using descriptors is that the bit-wise comparison can typically be implemented in a few machine instructions, in contrast to the q attribute value comparisons that are otherwise required.

3. Indexed Descriptor Files

A serious problem with the preceding algorithm is that there are as many descriptors as there are records. Thus, in very large data files, the number of disk accesses required to scan the descriptor file can be unacceptable [9]. A first improvement upon the algorithm is to associate each descriptor with several records rather than with a single record. Indeed, if the data records are stored as a file of *blocks* containing several records, it is quite natural to have a descriptor associated with all the records in the block.

Let β denote a block of p_β data records, $\{R_k\}$. (We let p_β, for *packing factor*, denote the number of records packed into block β.) We may create a descriptor, D_β, for the entire block of records by letting

$$D_\beta = \bigvee_{k=1}^{p_\beta} D_k \tag{3.1}$$

where $\bigvee_k D_k$ denotes the bit-wise (or logical) OR of the record descriptors of the block. Note that the ORing operation preserves Proposition 2.1 and its corollaries; if the query descriptor is not contained in the block descriptor, $Q \not\subseteq D_\beta$, then no record of the block can possibly satisfy the query, so it need not be accessed. Now we can modify our earlier retrieval procedure. Given a query descriptor Q, the procedure sequentially searches a file of block descriptors, called the first *index file*. If $Q \subseteq D_\beta$ then the block β is accessed and the actual attribute values $(v_1, \ldots, v_f)_R$ of each of its p_β records are compared with the query values $(v'_1, \ldots, v'_q)$. If the partial-match criteria is satisfied, this record R is added to the response set.

If we let r denote the size of the data file (total number of records R), and let p denote the *average* packing factor, then the index file of block descriptors contains $\lceil r/p \rceil$ descriptors. Each block descriptor must be accessed since the file of descriptors is being searched sequentially. To reduce the number of storage accesses required in this sequential search, it is natural to pack several of the block descriptors D_β into a single block of storage, especially since descriptors are short and of equal, fixed length. A descriptor for such a block α can be created by ORing each of the individual block descriptors D_β in α; that is,

$$D_\alpha = \bigvee_\beta D_\beta. \tag{3.2}$$

Readily, these derived descriptors may be accumulated in a second descriptor file. And we may modify the retrieval procedure to begin by sequentially searching this second, and very much smaller descriptor file. If $Q \subseteq D_\alpha$, then the block α in the first index file is accessed from storage and Q is compared with each D_β in α. If $Q \subseteq D_\beta$, then the block β of data records is accessed and the actual record attribute values are compared with the query attribute values.

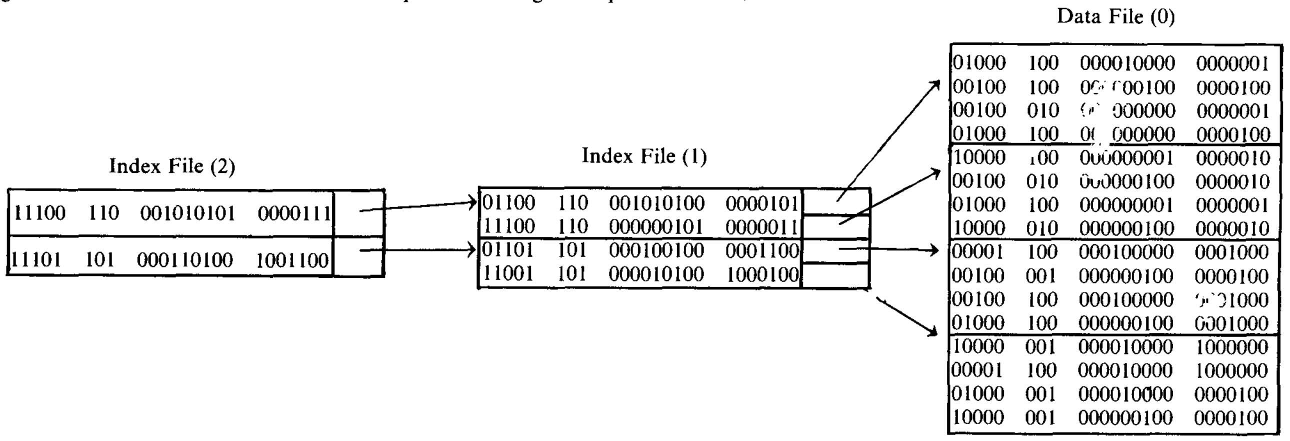
Of course, the descriptors D_α in the second descriptor file can themselves be blocked and ORed to form a third index file, and so on. Since each file of descriptors in this hierarchical structure serves as an index to the blocks of the lower level file, we call them *indexed descriptor files.* By convention, we let file(0) denote the file of data records; file(1) denote the index file of its block descriptors; file(2) denote the index file of its (file(1)) block descriptors, and so on. If d denotes the *depth* of the hierarchical structure, then index file(d) is the "highest" level of the structure and the only one that needs to be searched sequentially. We will parameterize our notation for descriptors, blocks, the number of records, and packing factors by file level, i. Thus we have $D(i)$, $\beta(i)$, $r(i)$, and $p(i)$.

Figure 2 illustrates a portion of an indexed descriptor file of depth 2. In this example, the data records are identified by four attributes which are represented by fields of width 5, 3, 9, and 7 in a 24-bit descriptor. The records in the data file(0) have been packed four records per block. The descriptors in index file(1) have been packed two descriptors per block. A more formal description of the retrieval procedure is given below.

```
procedure query ($v'_1, \ldots, v'_q$);
    | Given a set $\{v'_1, \ldots, v'_q\}$ of $q$ specified attribute values,
    | apply the transformations to form a query descriptor
    | with $q$ nonzero fields. |
    $Q \leftarrow \{T_j(v'_j)$:for specified query attributes, $j\}$;
    | Exhaustively search all descriptors in the highest level
    | index file($d$). |
    for each $D_\beta(d)$ in file($d$) do
        if $Q \subseteq D_\beta(d)$ then search ($\beta$, $d - 1$);
    end

recursive procedure search (beta, level);
    fetch block beta from storage;
    | Examine all records or descriptors in this block. |
    if level $> 0$
      then | This is a block of descriptors in an index file. |
          for each $D_\beta$(level) in block beta do
              if $Q \subseteq D_\beta$(level) then search ($\beta$, level $- 1$)
      else | This is a block of data records. |
          for each $R$ in block beta do
              if $(v_1, \ldots, v_f)$ satisfy $(v'_1, \ldots, v'_q)$
                then add $R$ to response set;
    end
```

Fig. 2. Portion of a Two-level Indexed Descriptor File Using Descriptors of Width, $w = 24$ bits.



To demonstrate the retrieval process, consider two queries in which only two query attribute values have been specified. Assume that the resulting query descriptors are:

Q_1 = 00001 000 000100000 0000000
Q_2 = 00000 010 000000000 0000010

Query 1 will access only the 2nd block of index file(1) and the 3rd block of the data file. Only the first record of this block (9th data record in the file) could possibly satisfy the query, although whether it does or not depends on the actual attribute values associated with the record and those specified in the query. Query 2 will access the first block of file(1) and the 2nd block of the data file(0). Either records 6 or 8 in the data file could satisfy the query.

Before continuing with a quantitative analysis of the retrieval procedure in the next section, we would like to make three comments regarding Figure 2 which is illustrative of the general procedure, but misleading concerning actual practice. First, a 24-bit descriptor is too small. In a very large database one expects to use descriptors with widths of 100 to 200 bits. Second, since descriptors are normally much shorter than the records of the data file, a packing factor, $p(1) = (2)$, for index file(1) is utterly unrealistic. Packing factors on the order of 100 are more normal. Third, the figure suggests that individual record descriptors are stored in the data file. This is unnecessary, and indeed, the search procedure never refers to a zero-level descriptor. The data file consists of just the data records themselves, with no additional stored information. The *concept* of a zero-level record descriptor is only useful in understanding the way data files are indexed, and index files created.

4. Analysis of Expected Accesses per Query

Let $\bar{t}_j(i)$ denote the expected (or average) number of 1-bits in the jth field, F_j, of a descriptor in file(i). Readily, $\bar{t}_j(i) \geq 1$. We begin by considering the probability that a given query descriptor Q is contained in (satisfied by) a descriptor $D(d)$ in the highest level index file(d). Assuming a uniform distribution of the transformed attribute values $T_j(v_j)$ (not of the attribute values v_j themselves), the probability that $F_j(Q)$ is contained in the jth field of $D(d)$ is simply expressed by $\bar{t}_j(d)/w_j$, which is the expected bit density in $F_j(d)$. Then the probability of a match in file(d) is simply

$$pr(Q \subseteq D(d)) = \prod_{j \in Q} \bar{t}_j(d)/w_j. \tag{4.1}$$

Consequently, the expected number of blocks accessed in file($d - 1$) is the number of descriptors examined in file(d) times the probability that any examined descriptor satisfies the query descriptor Q, or

$$\begin{aligned} E(\text{Blocks accessed in file}(d-1) \mid Q) &= \bar{b}(d-1) \\ &= r(d) \cdot pr(Q \subseteq D(d)) \\ &= r(d) \cdot \prod_{j \in Q} \bar{t}_j(d)/w_j \end{aligned} \tag{4.2}$$

where we are using $\bar{b}(d - 1)$ to compactly denote the expected number of blocks accessed in file($d - 1$).

Now consider a descriptor $D(d - 1)$ in a block β of file($d - 1$) that has been accessed in the query procedure. We need the conditional probability that $Q \subseteq D(d - 1)$ *given* that $Q \subseteq D_\beta(d)$. Since $Q \subseteq D_\beta(d)$, all of the 1-bits of Q are known to match 1-bits in $D_\beta(d)$; hence, the "effective" width of any F_j in $D(d - 1)$ is $\bar{t}_j(d)$. Consequently,

$$\begin{aligned} pr(Q \subseteq D(d-1) \mid Q \subseteq D_\beta(d)) \\ = \prod_{j \in Q} \bar{t}_j(d-1)/\bar{t}_j(d). \end{aligned} \tag{4.3}$$

Now the expected number of blocks accessed in file ($d - 2$) is the number of descriptors examined in file($d - 1$) (that is, the number of blocks accessed times the average packing factor $p(d - 1)$ for that file) times the probability of a match for each such record examined. Thus,

$$\begin{aligned} E(\text{blocks accessed in file}(d-2) \mid Q) &= \bar{b}(d-2) \\ &= \bar{b}(d-1) \cdot p(d-1) \cdot \prod_{j \in Q} \bar{t}_j(d-1)/\bar{t}_j(d) \\ &= [r(d) \cdot \prod_{j \in Q} \bar{t}_j(d)/w_j] \cdot p(d-1) \cdot \prod_{j \in Q} \bar{t}(d-1)/\bar{t}_j(d) \\ &= r(d) \cdot p(d-1) \cdot \prod_{j \in Q} \bar{t}_j(d-1)/w_j. \end{aligned} \tag{4.4}$$

But since each descriptor in file(d) corresponds to a block in file($d - 1$), we have

$$r(d) \cdot p(d-1) = r(d-1) \tag{4.5}$$

so,

$$E(\text{blocks accessed in file}(d-2) \mid Q) = \bar{b}(d-2) = r(d-1) \prod_{j \in Q} \bar{t}_j(d-1)/w_j. \tag{4.6}$$

In a similar manner it is easy to show that, in general,

$$E(\text{blocks accessed in file}(i) \mid Q) = \bar{b}(i) = r(i+1) \prod_{j \in Q} \bar{t}_j(i+1)/w_j \tag{4.7}$$

so that the total number of blocks accessed in the course of retrieving records satisfying a query Q is given by

$$E(\text{blocks accessed} \mid Q) = \Sigma_{i=0}^{d-1} \bar{b}(i) = \Sigma_{i=0}^{d-1} r(i+1) \cdot \prod_{j \in Q} \bar{t}_j(i+1)/w_j. \tag{4.8}$$

In this expression we have assumed that the highest level index file(d) is core resident so that no blocks need be accessed to sequentially examine it. If this is not the case, then a constant term, $\bar{b}(d) = r(d)/p(d)$, must be added to the expression above, since every block of file(d) must be accessed.

We can get a feeling for the implications of expression (4.8) by evaluating it for some representative numbers. Let the data file consist of $r(0) = 1{,}440{,}000$ records. Associated with these records will be 7 identifying attributes, so that each descriptor will consist of 7 fields. Each of these we make of width $w_j = 10$ bits, so that $w = 70$ bits. If the 1.44 million records of the data file are blocked with 24 records per block, there will be $r(1) = 60{,}000$ descriptors in index file(1). If these are in turn blocked with 128 descriptors per block, then there will be $r(2) = 470$ descriptors in index file(2). Five hundred descriptors can reasonably reside in primary storage, so file(2) can be exhaustively searched with no disk accesses.

Table I presents the average number of 1-bits in each of the descriptor fields of the two index files. If all 7 attributes (or descriptor fields) are specified in the query, then $\Pi_{j=1}^{7} \bar{t}_j(2)/w_j = .00249$, and $\Pi_{j=1}^{7} \bar{t}_j(1)/w_j = .0000544$. Substituting these into expressions (4.7) and (4.8) we have

$E(\text{blocks accessed in file}(1) \mid Q) = 1.172,$
$E(\text{blocks accessed in file}(0) \mid Q) = 3.264$

and, therefore

$E(\text{blocks accessed, or disk accesses} \mid Q) = 3.548.$

This example was *not* created for this paper! E. Cagley (who first proposed this kind of retrieval algorithm in [3]) and his associates in the Mathematics and Computation Laboratory of the Federal Preparedness Agency have created an indexed descriptor file to retrieve information from a file of census data of this magnitude [4]. The descriptor widths (2 machine words), packing factors, and bit densities are those observed in their system. In the two years that the implementation of this dynamic file has been operational, it has been observed that an average of between 3 and 4 disk accesses is sufficient to respond to a query for a singly fully specified record.

The preceding example assumes that all seven fields are specified in a query. It will be illustrative to examine the behavior of the system if just three of the fields are specified. In this case there will be approximately 1,440 records (technically buckets) that may satisfy the query specifications. Given the uneven distribution of 1-bits in the fields of the index descriptors, a query specifying fields 1, 2, and 3 represents a "best case" situation, while the specification of fields 5, 6, 7 represents a "worst case" situation. We will examine both, even though the way these index files were created [4] optimizes retrieval on the first three fields at the expense of retrieval using the last three fields (which, in fact, have never been used as a query). $\Pi_{j=1}^{3} \bar{t}_j(2)/w_j = .00396$, and $\Pi_{j=1}^{3} \bar{t}_j(1)/w_i = .001$. Thus, in the best case, it is expected that $1.861 + 60 = 61.861$ disk accessed must be made to retrieve approximately 1,440 records. In the worst case query. $\Pi_{j=5}^{7} \bar{t}_j(2)/w_j = .874$ and $\Pi_{j=5}^{7} \bar{t}_j(1)/w_j = .259$. Now the expected number of disk accesses to respond to the query is $411.09 + 1555.2 = 1966.29$ to retrieve the same number of records.

While expression (4.8) gives the exact expected number of disk accesses, it is not always an easy one to use in practice since it presumes a knowledge of which attribute fields have been specified in the query in order to evaluate the right-hand product. Frequently, only the total number, q, of attributes specifed in Q is known. We may simplify expressions (4.7) and (4.8) by *assuming* that the bit density in any single field is nearly the same as the overall *average bit density* $\overline{bd}(i)$ of the entire descriptor D. That is, we assume that

$$\bar{t}_j(i)/w_j \approx \overline{bd}(i) = \frac{\text{expected bits in } D(i)}{\text{width of } D} < 1.0$$

for all fields F_j in any index file(i). While this assumption of the relative invariance of bit densities per field is dependent on the particular method of creating the file structure, and may thus be invalid (as in the preceding example), we have found that the approximation often yields surprisingly accurate predicted access costs. Fur-

Table I. Average number of 1-bits in descriptor fields, F_j of two index files. (Note: The uneven distribution of bits in these fields is a characteristic of the particular way this file was created.)

F_j	$\bar{t}_j(2)$	$\bar{t}_j(1)$
1	1.0	1.0
2	1.1	1.0
3	3.6	1.0
4	7.2	2.1
5	9.3	3.7
6	9.5	7.3
7	9.9	9.6

thermore, the simplified form highlights important features of retrieval using indexed descriptor structures. And this form is more amenable to standard optimization techniques. Using this approximation, expression (4.7) becomes

$$E(\text{blocks accessed in file}(i) \mid q) \approx r(i+1) \cdot \overline{bd}(i+1)^q \tag{4.9}$$

and expression (4.8) becomes

$$E(\text{blocks accessed} \mid q) \approx \Sigma_{i=0}^{d-1} [r(i+1) \cdot \overline{bd}(i+1)^q]. \tag{4.10}$$

Since the density of 1-bits in the descriptors of index file$(i+1)$, $\overline{bd}(i+1) < 1.0$, the factor on the right decreases, often dramatically, as q increases. Consequently, as the number of attribute values specified in a query increases, the cost of retrieval *decreases*—in contrast to standard inverted list retrieval in which the cost actually *increases.*

From expression (4.10), it is seen that the expected number of blocks accessed when responding to a query is not only a function of the number q of query attributes specified; but it is also a function of the size, $r(i)$, and average bit density, $\overline{bd}(i)$, of each index file(i). Readily, one seeks to minimize retrieval cost by minimizing $r(i)$ (that is, increasing $p(i-1)$) and by minimizing the bit densities $\overline{bd}(i)$. Unfortunately, we should emphasize that the twin goals of both increasing the average packing factor while decreasing the average bit density are normally incompatible. As more descriptors are ORed to form a block descriptor, the number of bits in that block descriptor tends to increase. Controlling and optimizing these two parameters is just one of the interesting design problems associated with indexed descriptor files.

5. File Creation, Record Entry

The nature of the retrieval process using indexed descriptor files is independent of the particular way the file may have been created, even though its actual performance may be significantly affected by parameters established by the creation algorithm, that is, average bits in field-j of file(i), $\bar{t}_j(i)$, and the number of records per index file, $r(i)$. There is no "single" way to create indexed descriptor files. Indeed, this file structure admits several creation and/or entry procedures which may be selected, and then tuned for a particular application. These procedures fall into two broad classes which we call "bottom-up" and "top-down" file creation processes.

In "bottom-up" file creation, the data file is first organized by any desired method (it is not a function of the indexed descriptor file structure). For example, the records of the file described in Section 4, and in Table I, were first sorted lexicographically on the subfields of their descriptors. Then the first $p(0)$ data records are packed into a single block and its block descriptor formed to become the first record in index file(1). The next $p(0)$ data records are packed into a block, its block descriptor formed, and added to index file(1), and so on. The first $p(1)$ records (block descriptors) in index file(1) are packed into a single block and its block descriptor formed to become the first record of index file(2), and so on.

Readily, this method of file creation is optimal with respect to storage overhead, since every storage block is fully packed, except possibly the last block in any file. And given that the file has already been organized in some fashion, it is fast, requiring only a single sequential pass through the data file ORing and creating block descriptors as it goes. Readily, this method of file creation is best for static files, but it can be appropriate in the case of moderately dynamic files using a record insertion algorithm which simply enters the new record into the last partially filled block, and ORs its descriptor to each "covering" block descriptor. Since this latter portion of the file is randomly organized (by order of record insertion), the entire data file may be periodically reorganized and a new indexed descriptor file structure created over it.

Highly dynamic files are more easily maintained using "top-down" record entry. Here we assume an existing file structure with only partially filled blocks. Using the descriptor of a new record, the entry procedure searches the tree-structured index file to find the "best" data block in which to insert this new record—if a "best" block exists. The concept of "best" is normally a heuristic one which may be based on a large number of criteria, such as the number of new bits added to a block descriptor, or the Hamming distance between the old and new block descriptors. Similarly, several search algorithms may be used, such as depth-first, breadth-first, or ordered search. In all "top-down" entry procedures, the new record descriptor D is compared with the block descriptors in the highest level file(d). Each time a decision is made whether or not the associated block in file$(d-1)$ should be accessed and D compared with its descriptors, and so on. Both the adequacy of the resulting data organization and the efficiency of the entry procedure (in terms of storage accesses) are highly dependent on the particular parameters employed in the search process and its decisions. Some of the subtle interactions are too involved to discuss adequately in this paper.

It is apparent that each block descriptor in an index file "describes" a subcluster, cluster, or supercluster of records in the data file, depending on its level in the index. With "bottom-up" creation these descriptors merely reflect the clustered organization imposed upon the data file a priori. In "top-down" creation, the records of the data file are clustered "on the fly." Again, the actual clustering is determined by the particular search strategies and decision heurisms employed. Still the descriptor files form a useful vehicle for recording the existing structure on which decisions are based, and for conceptually studying (in retrospect) various clustering phenomena, such as cluster overlap.

Record deletion is nearly trivial in indexed descriptor files. The record is simply deleted from its data block; a new block descriptor is recalculated, with covering block descriptors rewritten and recalculated as necessary. At most two or three storage accesses are involved.

6. Conclusions

As shown in the preceding sections, partial-match retrieval using indexed descriptor files is certainly competitive, in terms of storage accesses, with the traditional inverted file technique; and when several attributes are specified in the partial-match query, it is superior.

We may also be concerned with the issue of storage overhead and its associated cost. This is a difficult problem to analyze precisely since it is dependent on the length of the data records, the width of their descriptors, the packing factors used in the various files, etc. However, we may observe that no additional information need be added to the data file (one need not store the block descriptor in its block); the only storage overhead is associated with the index files themselves. Given a reasonable packing of the data file, say $p(0) > 10$, then one relatively short descriptor record is inserted into the first index file for each 10 or more data records. Higher level index files may be even more highly compressed. Thus, with a "bottom-up" descriptor index we may expect storage overhead costs of less than 10 percent of the total data file. Cagley reports a typical storage overhead of 5 percent in his applications [4].

In index files that have been created by a "top-down" entry procedure which uses a heuristically guided process to "cluster" records and thus reduce descriptor bit densities, we do not fully pack the blocks in the data file so room will be left for later insertions. In this case we have observed storage overhead costs of from 20 to 40 percent of the data file, depending on how finely tuned the entry procedure is.

The issue of "tuning" an indexed descriptor system is an interesting one—precisely because it is possible to do so. Consider the problem of a record attribute which is seldom used in retrieval, but may be important when it is specified. With inverted files it is an all or nothing decision; either that attribute is inverted (with the same attendant costs as with inverting any other attribute) or it is not (forcing a sequential search of the entire file for retrieval on this attribute). In these files, the designer may decide on the relative importance of any attribute and allocate a proportional number of bits w in the descriptor. Both [1, 2] give formulae for optimally setting the dimensions of a bucket space, which are suggestive, but not quite accurate in this situation since they are based on a different expression for the expected number of storage accesses per query. By the same token, one may tune the system by using a deeper structure of three or more index files; this seems, however, to be unnecessary with files of less than a million records.

Received 3/79; revised 2/80; accepted 4/80

References

1. Aho, A.V., and Ullman, J.D. Optimal partial-match retrieval when fields are independently specified. *Trans. Database Systs. 4*, 2 (June 1979), 168–179.
2. Berman, W.J., and Pfaltz, J.L. Multi-dimensional bucket arrays. DAMACS Tech. Rep. TR-16-78, Univ. of Virginia, March 1978.
3. Cagley, E.M. A retrieval strategy for large, multi-key files requiring frequent updating. TR-75, Executive Office of the President (Office of Emergency Preparedness), Dec. 1971.
4. Cagley, E.M., et al. Information Management System Reference Manual. GSA/FPA/MCL TM-208, Oct. 1976.
5. Cardenas, A.F. Analysis and performance of inverted data base structures, *Comm. ACM. 18*, 5 (May 1975), 253–263.
6. DATAPRO Res. Corp. A Buyer's Guide to Data Base Management Systems, 1974.
7. Files, J.R., and Huskey, H.D. An information retrieval system based on superimposed coding. Proc. AFIPS Fall Joint Comptr. Conf., Vol. 35, AFIPS Press, Arlington, Va., 1969.
8. Knuth, D.E. *The Art of Computer Programming, Vol. 3.* Addison-Wesley, Reading, Mass., 1973.
9. Lefkovitz, D. The large data base file structure dilemma. Rep. 76-5, Univ. of Penn. Moore School, 1976.
10. Rivest, R.L. Partial-match retrieval algorithms. *SIAM J. Comptng. 5*, 1 (March 1976), 19–50.
11. Roberts, C.S. Partial-match retrieval via the method of superimposed codes. Proc. IEEE, Vol. 67, No. 12 (Dec. 1979), pp. 1624–1642.
12. Schroeder, J., et al. Stanford's Generalized Data Base System. Internat. Conf. on Very Large Data Bases, Framingham, Mass., Sept. 1975.
13. Severance, D.G., and Carlis, J.V. A practical approach to selecting record access paths. *Comptng. Surveys 9*, 4 (Dec. 1977), 259–272.
14. Vallarino, O. On the use of bit maps for multiple key retrieval. *ACM SIGPLAN Notices 11* (March 1976), 108–114.
15. Wiederhold, G. *Database Design.* McGraw-Hill, N.Y., 1977.

Corrigendum. Reports and Articles

Robert F. Ling, "General Considerations on the Design of an Interactive System for Data Analysis," *Comm. ACM 23*, 3 (March 1980), 147–154.

In the Acknowledgments section on page 153, the author expressed special thanks to one of the Associate Editors. Through an unfortunate error the wrong Associate Editor was named. The correct editor should have been Michael Marcotty. The error is deeply regretted.