A Superimposed Coding Scheme Based on Multiple Block Descriptor Files for Indexing Very Large Data Bases

A. Kent[1] R. Sacks-Davis[1] K. Ramamohanarao[2]

[1] *Department of Computer Science, Royal Melbourne Institute of Technology, Melbourne, Victoria 3000.*
[2] *Department of Computer Science, University of Melbourne, Parkville, Victoria 3052.*

Abstract

A new signature file method for accessing information from large data files containing both formatted and free text data is presented. The new method, called the multi-organizational scheme is proposed for indexing very large data files containing hundreds of thousands or possibly millions of records.

1. Introduction

For applications such as library systems, medical records systems and office automation, it is necessary to have access to large amounts of data. This data is stored in records (or documents) containing both formatted fields as well as free text. In order to efficiently retrieve information from such data bases, efficient access methods are required. A widely advocated method for indexing both formatted data and free text is the signature file method [15]. In this paper, a new method based on signature files, is presented for indexing large data files.

In the signature file method, a descriptor (or signature) is associated with each record, the descriptor being an encoding of the index terms used to retrieve the record. When a query is processed, the file of descriptors, rather than the data records, is examined for potential matches. Signature file methods have good retrieval properties and are storage efficient [9].

In order to form a signature, the terms used to index a record are mapped on to bit positions, and the corresponding bits in the record descriptor are set. Signature file methods provide the database designer with a good deal of flexibility in choosing the encoding method used to form the descriptors. Tradeoffs between storage efficiency, query times and insertion costs can be made by appropriate choices of the number of bits to be set per indexed term and the width of the record descriptors [23]. As well as setting bits for individual terms, it is possible to set bits using combinations of terms or substrings of terms. This property makes the signature file method particularly suitable for applications involving free text. It is also possible to encode hierarchies of terms, such as tree structures. As a result, signature file methods have been proposed for applications for which these structures occur, such as Prolog databases. Signature file methods have been proposed for multikey retrieval [20–22], text retrieval [9,10,13], Prolog systems [8,19,25,26], office systems [6,11,18], statistical databases [27] and filtering methods [2,4,16]. Signature file methods are well suited to hardware implementation [1,3].

In order to efficiently access large files, a number of strategies can be adopted for signature files. These include

(i) the use of multilevel indexes, so that signatures are formed for blocks of records as well as single records [17,21,23],

(ii) the incorporation of special storage representations, referred to as bit slice implementations, to reduce the amount of storage that has to be processed on query [20,27],

(iii) the use of special encoding methods which use the frequency of occurrence of the terms to be indexed in determining the number of bits to be set per term [10,23], and

(iv) the use of compression techniques on the signatures to reduce the size of the signature file [11].

A method that uses the first three of these approaches has been proposed in [23]. Two levels of index are supported: block descriptors are formed for blocks of records and record descriptors are maintained for individual records. The file of block descriptors is stored using the bit slice technique. In the method proposed in [23], the terms in the database which occur most frequently (referred to as common terms) are identified, and bits are set in the block descriptor file for *pairs of common terms* as well as for individual terms. The method was shown to be effective for large data files.

In this paper a new signature file method, suitable for large data files, is proposed. Like the method proposed in [23], it uses descriptors formed for blocks of records. However, rather than using a single block descriptor file, the new

method uses multiple block descriptor files. To form one of these block descriptor files, records are logically grouped into blocks, but the mapping function used to group records can differ for each block descriptor file. Hence the method is based on multiple logical organizations of the records in the data file and is referred to as the *multi-organizational scheme*. Both theoretical and experimental results are presented in order to demonstrate that the method is efficient.

In the following section a brief description of signature file methods is presented. A description of the multi-organizational scheme is then provided. Alternative methods for evaluating a query using the multi-organizational scheme are presented in Section 4. In order for the proposed method to be effective, it is necessary to identify the frequently occurring terms, called common terms, and treat these terms differently from the other terms. A technique for handling common terms is presented in Section 5.

A detailed description and analysis of the method are presented in an extended version of this paper [14]. The analysis presented in [14] is based on a mathematical model that assumes that the data are Zipf distributed. Some sample results obtained using this model are presented in Section 6. The conclusions are presented in Section 7.

2. Signature Files

A signature is a bit string formed from term values that are used to index a record. Indexing using signature files assigns a signature or descriptor to every record in the data file. To perform a query, the descriptors are examined to identify potential matches. For example, to form a descriptor for a record using the method of superimposed coding, each term in the record to be indexed is identified, and a descriptor is formed for that term (called a term descriptor). These term descriptors are formed by using a hash function to convert the term values into a bit string of length b with exactly k bits set to 1 and $b-k$ bits set to 0. A record descriptor, which describes the entire contents of the record, is formed by superimposing (inclusive ORing) the term descriptors. To perform a query, the file containing the record descriptors is examined to determine potential matches, and then the data file is accessed to retrieve the data.

An example of superimposed coding is given below.

John	0000 0101
Smith	0100 1000
Record Descriptor	0100 1101

This method naturally supports variable numbers of terms per record as the number of terms does not affect the descriptor length. Multi-valued fields (fields in a record which can contain more than one value) and free text are also handled easily as there is no distinction made between values belonging to the same field (in contrast with methods which concatenate rather than superimpose term descriptors). In fact, to ensure that the same values in different fields do not match, a different hash function is required per field (in practice, this can simply involve adding the field number to the seed used by the hash function).

To check if a descriptor matches a query, the query terms are also hashed to form a query descriptor using exactly the same method used to generate the record descriptors. If every bit set in the query descriptor is also set in the record descriptor, then the record descriptor is a potential match.

Superimposed coding can result in cases where the query and record descriptors match, but the record does not actually contain the query terms. This is called a false match. The probability of a false match occurring is a function of the number of bits set in the query and record descriptors. False matches place a practical restriction on the number of terms that a record may contain for a given descriptor size.

Detailed formulas for calculating false match probabilities and values for k and b are presented in [10,20,24]. It has been shown that the optimal bit density in a descriptor (to minimize the probability of a false match for a given number of available bit positions) is 50%. This usually results in a high number of bits that must be checked in the descriptors on query. In certain applications it is better to use a lower than optimal density, particularly when using the bit sliced descriptor file (described below), to reduce the number of bit positions that need to be checked.

There are two common ways of storing descriptors in a file. These are referred to as the bit string and bit slice methods respectively. The bit string approach [10,12] stores the descriptors sequentially in the descriptor file. Pointers to the corresponding records can be stored with the descriptors. This approach has the advantage of simplicity, especially for update and insertion operations. One problem is that for large files, queries can become slow as the whole descriptor file must be read to retrieve and examine the descriptors. This method can work well for relatively large files if queries are batched [5].

Instead of storing the descriptors as a file of N (where N is the number of records in the file) bit strings each consisting of b bits, the bit slice method [8,20,27] stores the descriptors as b N bit long bit slices (see below).

000001	00 . . .
001010	00 . . .
. . .	01 . . .
. . .	00 . . .
. . .	01 . . .
. . .	10 . . .
Bit String File Organization	Bit Slice File Organization

Because only a subset of the bit positions in the record descriptors needs to be examined on query, only a fraction

of the descriptor file needs to be retrieved on query. A single seek and an N bit long read operation will retrieve a bit vector which identifies which record descriptors have a particular bit position set. The bit slice approach has some disadvantages – expanding the capacity of the index may require rebuilding the whole index and interactive insertions require approximately one disk read and write operation per bit position to be set. Also, pointers to records can no longer be stored with the descriptors. Instead a separate pointer file is needed.

For very large data files, queries using the bit slice approach can still be expensive as bit slices of length N bits must be read from disk. For example, for a database containing 200,000 records, the slices will be 25,000 bytes long or approximately 49 × 512 byte pages. One way to reduce the number of bits that need to be examined is to reduce k, the number of bits set per term value. The main disadvantage of this approach is the corresponding increase in b (required to keep the number of false matches constant) which increases the index file size. In practice this means the bit slice method is less storage efficient than the bit string approach.

A method that uses two levels of index has been proposed in [21] and later refined and analyzed in [22,23]. Rather than using just a single file of record descriptors to locate matching records, a higher level of index (called the block descriptor file) is formed. With this approach, records are allocated to blocks and signatures, referred to as block descriptors, are formed for each block. As in the one level schemes, described above, record descriptors are also formed for individual records. If N_s is the number of blocks and N_r is the number of records per block, then $N = N_s \cdot N_r$ and the number of terms per block will be $N_r \cdot s$ where s is the number of terms per record. Since the number of terms per block is much greater than the number of terms per record, the block descriptors will be much larger than the record descriptors. Typical parameter values can be found in [21–23]. The block level index is first examined on query to identify which blocks of records match, and then the record descriptors of records from the matching blocks are examined to identify the individual matching records. The block descriptor file is stored as a bit sliced file to facilitate efficient query processing while the record descriptors are stored in the bit string format. The block size is selected so that the record descriptors for a block will all fit in a single, or small multiple of, disk pages.

The two level scheme has been shown to perform well but does suffer from problems with certain multi-term queries [7]. When multiple terms are specified in a query, there may exist blocks that contain all of the terms, but not within a single record. At query time the existence of such blocks lead to what we call *unsuccessful block matches*. Unsuccessful block matches are more likely when the values specified in a multi-term query are common, that is they appear in many records. The problem can be minimized by reducing the number of records per block, or by using special encoding schemes [23].

In [23] an encoding scheme is presented for which bits are set in the block descriptors for pairs of common terms as well as for single terms. The extra bits that are set are referred to as combination bits and do not significantly add to the storage overhead. This is because the number of combination bits set for a pair of common words will generally be much less than k. The presence of these combination bits significantly reduces the number of unsuccessful block matches and makes the two level scheme efficient for multi-term queries.

In [23] it is also demonstrated that setting extra bits can also be used to make the signature file method very effective for free text retrieval. By setting a small number of bits for pairs of adjacent words in text, the signature file method provides direct support for queries which specify word phrases as well as single words at low cost.

If combination bits are not used, unsuccessful block matches occur in the two level scheme since the block descriptor file only identifies blocks that contain matching records rather than the individual matching records. If the first level of index examined could identify individual records, then unsuccessful block matches would no longer be a problem. The record descriptor file also could be eliminated, although a pointer file to map record numbers to data file pointers may still be required. In the multi-organizational approach described in the following section, it is advocated that instead of forming block descriptors by setting k bit positions per term, k separate block descriptors are formed, each having only one bit set per term. These block descriptors are then stored in k logically separate block descriptor files. In the two level scheme, records are assigned to blocks using the mapping function $block\# = record\#\ div\ block\_size$. In the multi-organization scheme each record number is mapped to a block number using a possibly different mapping for each of the k block descriptor files. Each of the files with a different mapping function is referred to as an organization. The k sets of matching blocks from a query can now be used to identify individual matching records. The method is described in the next section.

3. Multi-Organizational Scheme

In a two level scheme, based on a single block descriptor file, descriptors are formed for blocks of records. A query is answered by first determining which blocks contain all of the terms specified in the query and then for each matching block, the record descriptors representing records contained within that block are retrieved and examined for possible matches.

One way to view the organization of the two level scheme is to associate a logical record number, i, $i = 0,1,...,N-1$, with each record and a mapping

$$block(i) = i\ div\ N_r$$

which determines the block, $block(i)$, containing record i.

If a particular block, j say, matches a query, then a reverse mapping is performed to determine that records $j \cdot N_r$ through $(j+1) \cdot N_r - 1$ are potential record matches. The actual record matches are then determined by retrieving the record descriptors for each of these records.

Rather than having one block descriptor file with a single record-to-block mapping function, consider a scheme with several block descriptor files, each with a possibly different record to block mapping. The way records are allocated to blocks when forming a block descriptor file will be referred to as an organization of the collection of records.

For the j^{th} block descriptor file, the record to block mapping function will be written as

$$block_j(i) = r_j(i) \; div \; N_r$$

where $r_j(i)$: $[0,N-1] \rightarrow [0,N-1]$ maps each record number, i, to a new record number $r_j(i)$. In order for a particular j that the values $r_j(i)$ be unique, the following simple mapping can be used

$$r_j(i) = (i \cdot P_j) \; mod \; N$$

P_j must be chosen so that the greatest common denominator of P_j and N is 1. Typically, P_j is chosen to be a prime number greater than N_r. In the case $P_j = 1$, the mapping becomes the identity function $r_j(i) = i$. Note that it is assumed that an estimate, N, of the capacity of the database is required when the system is initialized. This issue is further considered in Section 7 where a modified mapping function is presented. Each of the block descriptor files is formed so that the m^{th} block descriptor in the j^{th} file, $m = 0,1,...,N_s-1$, is formed from all the records, i, for which $block_j(i) = m$.

This method is illustrated by the example in Figure 1.

$block_2(i)$	$r_2(i)$	i
0	0	0
	1	13
	2	10
	3	7
1	4	4
	5	1
	6	14
	7	11
2	8	8
	9	5
	10	2
	11	15
3	12	12
	13	9
	14	6
	15	3

Organization 2
$P_2 = 5$

$block_1(i)$	$r_1(i)$	i
0	0	0
	1	1
	2	2
	3	3
1	4	4
	5	5
	6	6
	7	7
2	8	8
	9	9
	10	10
	11	11
3	12	12
	13	13
	14	14
	15	15

Organization 1
$P_1 = 1$

Figure 1: Example of multi-organizational scheme with two organizations

In Figure 1 there are two block descriptor files with two different organizations for a database of 16 records (N = 16). In each file, block descriptors are formed for groups of 4 records (N_r = 4) and there are 4 block descriptors in each file (N_s = 4). In the first organization, the block containing record i is computed using the record mapping function

$$r_1(i) = i$$

and in the second file

$$r_2(i) = (i \cdot 5) \; mod \; N$$

Now consider a query which specifies a single term that is contained only in record 10. Assuming there are no false block matches, it can be determined that block 2 of organization 1 and block 0 of organization 2 are the only blocks which satisfy the query. Observe that the only record in both block 2 of organization 1 and block 0 of organization 2 is record 10. Thus it can be determined, in this case, which records satisfy the query using only information about the block matches. A very important property of the multi-organization method is that no record descriptors need be maintained and only block descriptors are stored.

Now suppose that another query is specified and that records 10 and 14 contain the query terms. In this case blocks 2 and 3 of organization 1 and blocks 0 and 1 of organization 2 are matching blocks. By using information about the matching blocks, it can be deduced that records 10, 11, 13 and 14 are potential record matches since these records belong to both matching blocks in each organization. Records 10 and 14 are true matches and records 11 and 13 are referred to as false matches. Obviously it is desirable to eliminate the occurrence of false record matches. Several strategies can be used to restrict the number of false matches that can occur. The number of false matches will tend to be large for those queries for which there are a large number of matching blocks in each organization. In our example, the query term occurred in 50% of the blocks in each organization. Terms that occur in a large number of blocks will be referred to as common words. In order to restrict false matches, common words will be treated specially by the indexing method and only terms that are not common (referred to as *regular words*) will be indexed in the way described above.

False matches can also occur due to collision errors generated by the underlying hashing methods. The number of these false matches can be restricted by an appropriate choice of the descriptor size b.

A problem with using a two level scheme with just a single organization is that unsuccessful block matches will occur. For a query which specifies two terms, an unsuccessful block match will occur if a block contains the two terms in different records, r_X and r_Y, and the block contains no records with both terms. For a method with multiple organizations and multiple block descriptor files, unsuccessful block matches will diminish because although two records

r_X and r_Y may appear together in a single block of one organization, they will almost certainly belong to different blocks in one of the other organizations. It can be shown that provided the frequency of indexed terms is limited (i.e., only non-common terms are indexed), unsuccessful block matches can be virtually eliminated using multiple organizations.

The file structure for indexing regular words is illustrated in Figure 2.

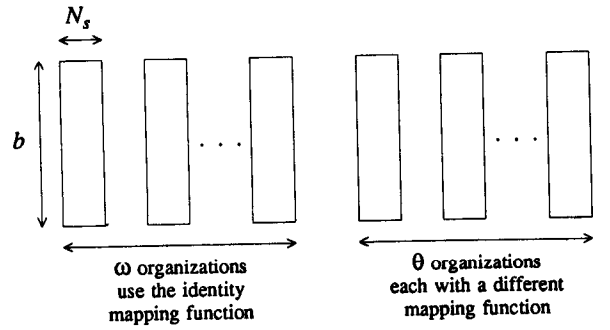

Figure 2: Multi-Organizational Scheme File Organization

There will be k block descriptor files where $k = \omega + \theta$. Each block descriptor file contains N_s descriptors of length b bits. The first ω organizations use the identity mapping function

$$r_j(i) = i, \quad j = 1,2, \ldots, \omega.$$

It will be seen that query evaluation is more efficient if there are a number of organizations with the identity mapping function. The hashing function used to map the terms into a bit position between 0 and $b-1$ will be different for each of the ω files that correspond to the identity organization. The processing of these files will simply involve the retrieval of the appropriate bit slices from each of the files and the ANDing of the resultant vectors. The remaining θ organizations each have a unique mapping function not equal to the identity function. These organizations are provided to eliminate unsuccessful block matches and identify actual record matches, thereby ensuring that queries which contain more than one term can be evaluated as efficiently as queries with a single term. For these organizations

$$r_j(i) = (i \cdot P_j) \bmod N, \quad j = \omega+1, \ldots, k.$$

In order to reduce the possibility that two blocks in different organizations contain more than one record in common, the values of P_j, $j = \omega+1, \ldots, k$ are chosen so that the minimum difference between any pair of distinct values P_j and P_l is N_r.

The indexing structure provided by the descriptor files displayed in Figure 2 will provide the logical record numbers of matching records as a result of a query. The data records will be stored in a separate file that must then be accessed. If the data records are of fixed length, then the physical address of a matching record may be computed from the logical record number. If, however, the data records are of variable length, then a separate file of pointers to the data records must be maintained. For large databases, this file of pointers must reside on disk. If this is the case, two disk accesses are required to retrieve a matching record once its logical address is determined.

4. Query Evaluation

In order to answer a query, a query descriptor is formed for each of the block descriptor files. For simplicity, it will be assumed that a single regular word has been supplied in the query. Queries which specify common words or multiple terms will be considered later. In the case of single term queries, there will be exactly one bit set in each of the query descriptors. The corresponding slices are retrieved from the block descriptor files and stored in memory. Since the first ω slices correspond to the identity organization, these slices can be ANDed together to form a single slice. At this stage, there are $\theta + 1$ slices of length N_s bits stored in memory. Each of these slices corresponds to a different organization. It is now required to determine the record matches using the information about the block matches for the $\theta + 1$ organizations. Two methods for determining the record matches are described below.

4.1. Expanding the Bit Slices

One approach is to transform each of the bit slices of length N_s into bit vectors of length N containing potential record matches. For the bit slice corresponding to the identity mapping, it is obvious that a bit set in the i^{th} position implies that records $i \cdot N_r$ through $(i+1) \cdot N_r - 1$ are potential record matches. Hence, when expanding the bit slice to a bit vector of length N containing the potential record matches for the identity organization, a bit set in the i^{th} position of the bit slice will set the bits $i \cdot N_r$ through $(i+1) \cdot N_r - 1$ in the bit vector. If the potential record matches are determined for each of the different organizations, then $\theta + 1$ bit vectors of length N containing potential record matches will be formed. The actual record matches can then be obtained by ANDing these bit vectors together to form a single vector of length N.

Unfortunately, for the θ organizations that use mappings other than the identity function, it is more difficult to obtain the potential record matches. Recall that the record to block mapping function is

$$block_j(i) = r_j(i) \ div \ N_r$$

where

$$r_j(i) = (i \cdot P_j) \bmod N.$$

In order to expand the bit slice to a vector containing potential record matches a reverse mapping is needed. The following algorithm maps a record number $r_j(i)$ in the j^{th} organization to the original record number i.

for $m = 0$ to $P_j - 1$
  if $(r_j(i) + m \cdot N) \ mod \ P_j = 0$ then
    $i = (r_j(i) + m \cdot N) \ div \ P_j$
    done
  endif
endfor

The problem with this algorithm is that it is not particularly efficient (due to the loop). The algorithm can be made more efficient if some values are precalculated and stored in a table. Initialization of the table is shown below.

for $i = 0$ to $P_j - 1$

$$table_j\left[\left(\left\lceil i \cdot N \ / \ P_j \right\rceil \cdot P_j\right) mod \ N\right] = i \cdot N$$

endfor

The reverse mapping function then becomes simply

$$i = (r_j(i) + table_j[r_j(i) \ mod \ P_j \ ]) \ div \ P_j$$

Unfortunately the implementation based on expanding the bit slices suffers a number of problems. Firstly, the internal buffers for storing the potential records matches are N bits in length. For large N, this will impose very heavy memory requirements. Also the cost of expanding the bit slices is large, even if the above procedure is used. For every matching block in each organization, the reverse mapping function must be invoked N_r times, once for every record in the block. The corresponding bit in the bit vector of length N must then be set. An implementation that does not require a reverse mapping is proposed in the next section.

4.2. A Non-Expanding Implementation

Rather than expand the retrieved bit slices of length N_s into vectors of length N, it is possible to locate matching records using the $\theta + 1$ bit slice buffers of length N_s. A search for matching records then involves an iteration through every possible record number from 0 to $N-1$. To test if a particular record matches the query terms, the block number of the record is calculated for each of the different organizations. Each of the $\theta + 1$ buffers is then tested to determine whether each of these blocks have matched the query. This may seem to require more calculations than the previous method. However, if the buffers are examined in a certain order, the computation time can be minimized.

First of all, it is not always necessary to calculate all $\theta + 1$ block numbers for a record. The calculation for a particular organization need only proceed if all the previous organizations matched the query for the particular record under examination. Since the first organization uses the identity mapping function, the mapping function does not need to be evaluated. If the average density of a bit slice is μ and since the first bit slice stored in memory is a result of ANDing together ω slices retrieved from the block descriptor files formed using the identity mapping function, the expected density of the first bit slice in memory will be μ^{ω}. This significantly reduces the total number of calculations. Note that for true matches, all θ block numbers must be calculated. For false matches however, on average only $\mu^{\omega} \cdot N$ calculations need to be performed for the first non-identity organization and only $\mu^{\omega+1} \cdot N$ for the second non-identity organization. The total number of times the mapping function, $r_j(i) = i \cdot P_j \ mod \ N$, is computed is then approximately

$$(\mu^{\omega} + \mu^{\omega+1} + \mu^{\omega+2} + \cdots + \mu^{\omega+\theta-1}) \cdot N + true \ matches \cdot \theta$$

$$\approx \frac{\mu^{\omega} \cdot N}{1 - \mu} + true \ matches \cdot \theta$$

Of course, a bit set to zero in any of the bit slices implies that all the records in the corresponding block can not satisfy the query. Since the first organization is based on the identity mapping, a bit set to zero in this organization implies that all the records in the corresponding block can be skipped by simply adding N_r to the current record number being tested. Similarly, if a complete word is found to be zero, then 32 blocks of records can be skipped (assuming a word size of 32 bits). Comparing a word to zero can significantly reduce the total CPU time required as a single word comparison is much faster than checking the 32 bits individually. It is therefore advantageous to choose ω sufficiently large so that the numbers of zero words will be high.

5. Common Words

As described previously, it is necessary to identify common words and treat them differently from the other terms that are used to index records in the database. Common words will be formally identified as follows. If there are N records in the database, each containing s terms, then there will be $N \cdot s$ terms in the database. Suppose that the number of distinct terms in the database is N_D. If these terms are labelled $v_1, v_2, \ldots, v_{N_D}$ so that the subscript i refers to the rank of the terms, then v_1 will be the most commonly occurring term in the database, v_2 the second most commonly occurring term and so on. The C most commonly occurring terms, $v_1, v_2, \ldots, v_C$ will be referred to as common words. The value C will be a parameter of the indexing scheme. The other terms, $v_{C+1}, \ldots, v_{N_D}$, will be formally referred to as regular words.

One approach for creating an index to the data file for the common words would be to store a dedicated bit slice of length N bits for each common word. Then, for any particular common word, a bit set in the i^{th} position of the corresponding slice would indicate that the i^{th} record in the database contains that common word. Unfortunately, this approach becomes very expensive of storage for large data files for even a moderate number of common words. The approach taken to reduce storage costs is similar to the scheme used for indexing regular words.

For each common term, v_i, θ_i slices of length $N_{s,i}$ will be formed. Both θ_i and $N_{s,i}$ can vary from common word to common word. Each of the θ_i slices corresponds to a different organization of the database. Each organization con-

tains $N_{s,i}$ blocks of $N_{r,i}$ records where

$$N_{s,i} = \left\lceil \frac{N}{N_{r,i}} \right\rceil$$

A bit set in the j^{th} position, $j = 0,1, \ldots, N_{s,i}-1$ of a slice indicates that the j^{th} block in the corresponding organization contains v_i.

There is one implementation consideration when the value of N_s varies per common term. To identify matching records, buffers must be allocated to hold slices retrieved from disk. Slices formed with the same mapping function and the same value of N_s can be ANDed together into a single buffer resulting in a sparser slice. ANDing several slices into a single buffer is advantageous as sparse vectors make processing faster (false matches are identified more quickly) and less buffers need to be examined. This cannot be done however when different values of N_s are used. It is therefore advantageous to restrict the number of distinct $N_{s,i}$ values used for common words in order to efficiently answer a query which specifies a number of different common words.

It is possible to AND together slices formed using different values of N_s if the mapping function is changed to $block_j(i) = r_j(i) \, mod \, N_s$. If values of N_s are restricted such that all values are 2^n times the smallest N_s value used, then shorter slices can be ANDed with longer slices by appending multiple copies of the shorter slices to form a single longer slice (see Figure 3).

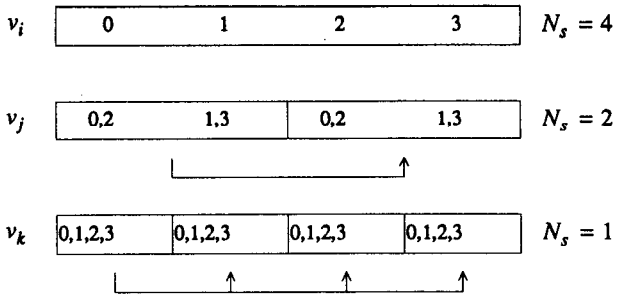

Figure 3: Example showing how short slices can be ANDed with longer slices (N=4)

6. Results

An analysis of the method is presented [14]. The analysis is based on a model for which it is assumed that the data is Zipf distributed [28], namely, p_i, the probability of occurrence of the i^{th} ranked term in the data file is α/i for some constant α. Some results obtained using this model are given in Table 1. Figure 4 contains a list of variables used in this table.

storage	storage overhead in bits per indexed term (total overhead in bits is *storage*·*s*·*N*)
N_r	number of records per block
N_s	number of blocks
ω	number of identity organizations
θ	number of non-identity organizations
C	number of common terms
μ	average bit density in a block descriptor file
fmr	number of false matches per true record match for a query specifying the $C+1^{th}$ ranked term.

Figure 4: List of symbols

Table 1 shows the effect of varying both the block size and the storage allocated per term value for a 2 million record database with 25 terms per record. The limit on the number of organizations, k, was constrained to be less than or equal to 12. For a system with 4K byte disk pages (32,768 bits), the N_s values presented all require one or more pages per slice. If the desired length of slices is to be 2 disk pages (N_s = 65,536) then an overhead of 40 bits per term occurrence is needed to keep the number of false matches to approximately 1 for every 25 true matches for the $C+1^{th}$ term. For single term queries, the $C+1^{th}$ term will provide the most false matches.

These results demonstrate the good perfomance can be obtained with this method at low storage overheads. An overhead of 40 bits per term occurrence compares favourably with the inverted file method which requires for each term occurrence a pointer, a (possibly shared) entry in a dictionary file where the term itself is stored and free space overheads in both the pointer and dictionary files, the size of which depend on the storage management algorithms used.

In order to understand the query perfomance consider, again, the entry for which the overhead per term is 40 bits and the block size is 32. Since false matches are rare, the cost of answering a query will be 11 disk seeks ($\omega+\theta = 11$) plus approximately 2 seeks for every matching record. One of these seeks is required to retrieve the data itself, and the other seek is required to retrieve a pointer to the data. If the file of 2 million records can be kept resident in memory, or if the data records are of fixed length, one seek can be avoided and the cost of answering a query reduces to approximately $11 + n$ seeks for n matching records.

The Table also demonstrates that the number of common words typically required will be small.

N = 2,097,152 s = 25							
storage	N_r	N_s	ω	θ	C	μ	*fmr*
40.0	64	32768	6	6	1449	0.309	0.1244
40.0	32	65536	5	6	448	0.258	0.0424
40.0	16	131072	5	6	374	0.258	0.0117
40.0	8	262144	5	7	140	0.258	0.0021
40.0	4	524288	5	7	113	0.258	0.0005
48.0	64	32768	5	7	1864	0.257	0.0125
48.0	32	65536	5	7	998	0.247	0.0046
48.0	16	131072	5	7	530	0.238	0.0016
48.0	8	262144	5	7	277	0.229	0.0006
48.0	4	524288	5	7	140	0.220	0.0002
56.0	64	32768	5	7	2308	0.220	0.0042
56.0	32	65536	4	7	705	0.185	0.0014
56.0	16	131072	4	7	513	0.185	0.0004
56.0	8	262144	4	8	142	0.185	0.0001
56.0	4	524288	4	8	104	0.184	0.0000
64.0	64	32768	4	8	2134	0.185	0.0004
64.0	32	65536	4	8	1028	0.179	0.0002
64.0	16	131072	4	8	454	0.172	0.0001

Table 1: Example parameters for a 2 million record database

In order to experimentally confirm the results prediced by the mathematical model, a practical system using a library database was implemented. The results are presented in [14] and are consistent with those predicted by the mathematical model.

7. Conclusions

The multi-organizational scheme has many properties which make it suitable for indexing very large data files. Query performance is good and storage overheads are low. Like other signature file methods based on superimposed coding schemes, indexing on word parts and word phrases can be supported at low cost, making the method suitable for indexing free text as well as formatted data.

Like the inverted file method, the cost of interactive insertions is high since a number of disk accesses must be performed per term per record. However, a fast batch insertion algorithm is proposed in [14] for which the cost of insertion is typically 2-4 disk accesses per record, irrespective of the number of terms per record.

The multi-organizational scheme is similar in many respects to the two level scheme presented in [23] but does have a number of advantages:

(i) The multi-organizational scheme performs better for multi-term queries as unsuccessful block matches are virtually eliminated.

(ii) The multi-organizational scheme performs better for large values of s as combination bits do not need to be set for pairs of common words, the number of which increases quadratically with the number of common terms per record.

(iii) The two level scheme requires the identification of far greater numbers of common words. This means a more detailed analysis of the record contents needs to be performed when a database is created and filters to identify the common words need to be designed.

(iv) No record descriptors are required and if fixed length records are used or if the pointer file can be stored in memory, a disk access per record retrieved can be saved with the multi-organizational scheme.

The multi-organizational scheme does require more CPU time to perform queries than the two level scheme due to the mapping functions. The additional cost can be controlled, however, by using a number (ω) of initial slices using the identity mapping function ($r(i) = i$) to reduce the number of times the mapping function needs to be evaluated.

8. References

1. M. W. Allen, Jayasooriay and R. M. Colomb, "Attached Index Machine: A New Form of Processor and its Application to Partial Match Data Retrieval", *Proceedings of the Ninth Australian Computer Science Conference*, Canberra, A.C.T. 2061, Australia, 29-31 January 1986, 347-355.
2. E. Babb, "Implementing a Relational Database by Means of Specialized Hardware", *ACM Transactions on Database Systems 4*, 1 (March, 1979), 1-29.
3. P. B. Berra, S. M. Chung and N. Hachem, "Computer Architecture for a Surrogate File to a Very Large Data/Knowledge Base", *IEEE Computer 20*, 3 (March 1987), 25-32.
4. B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors", *Comm. ACM 13*, 7 (July 1970), 422-426.
5. S. Christodoulakis, "Access Files for Batching Queries in Large Information Systems", *Proceedings of ICOD II*, Aug. 1983.
6. S. Christodoulakis, "Framework for the Development of an Experimental Mixed-Mode Message System", in *Research and development in information retrieval (ACM)*, C. J. Van-Rijsbergen (editor), Cambridge University Press, Cambridge, July 2-6, 1984, 1-20.
7. R. M. Colomb, "Use of Superimposed Code Words for Partial Match Data Retrieval", *Australian Computer Journal 17*, 4 (November 1985), 181-188.
8. R. M. Colomb and Jayasooriah, "A Clause Indexing System for PROLOG Based on Superimposed Coding", *Australian Computer Journal 18*, 1 (February 1986), 18-25.
9. C. Faloutsos, "Access Methods for Text", *ACM Computing Surveys 17*, 1 (March 1985), 49-74.
10. C. Faloutsos and S. Christodoulakis, "Design of a Signature File Method that Accounts for Non-Uniform

Occurrence and Query Frequencies", *Proceedings of 11th Conference on Very Large DataBases*, Stockholm, August 21-23, 1985, 165-170.

11. C. Faloutsos and S. Christodoulakis, "Description and Performance Analysis of Signature File Methods for Office Filing", *ACM Transactions on Office Information Systems 5*, 3 (July 1987), 237-257.
12. J. R. Files and H. D. Huskey, "An Information Retrieval System Based on Superimposed Coding", *Proceedings AFIPS, Fall Joint Computer Conference 35* (1969), 423-432, AFIPS Press.
13. M. C. Harrison, "Implementation of Substring Test by Hashing", *Comm. ACM 14* (1971), 777-779.
14. A. J. Kent, R. Sacks-Davis and K. Ramamohanarao, "A Signature File Scheme Based on Multiple Organisations for Indexing Very Large Databases", Tech. Rep. 87/5, Dept. of Comp. Sci., RMIT (submitted for publication), 1987.
15. D. E. Knuth, *The Art of Computer Programming, vol. 3: Sorting and Searching*, Addison-Wesley, Reading, Mass., 1973.
16. M. D. McIlroy, "Development of a Spelling List", *IEEE Transactions on Communications COM-30*, 1 (January 1982), 91-99.
17. J. L. Pfaltz, W. J. Berman and E. M. Cagley, "Partial-Match Retrieval Using Indexed Descriptor Files", *Comm. ACM 23*, 9 (September 1980), 522-528.
18. F. Rabitti and J. Zizka, "Evaluation of Access Methods to Text Documents in Office Systems", *Proceedings of the 3rd Joint ACM-BCS Symposium on Research and Development in Information Retrieval*, Cambridge, Mass., July 2-6, 1984, 21-40.
19. K. Ramamohanarao and J. Shepherd, "A Superimposed Codeword Indexing Scheme for Very Large Prolog Databases", *Proceedings of the Third International Conference on Logic Programming*, London, 1986, 569-576.
20. C. S. Roberts, "Partial-Match Retrieval via the Method of Superimposed Codes", *Proceedings of the IEEE 67*, 12 (December 1979), 1624-1642.
21. R. Sacks-Davis and K. Ramamohanarao, "A Two Level Superimposed Coding Scheme for Partial Match Retrieval", *Information Systems 8*, 4 (1983), 273-280.
22. R. Sacks-Davis, "Performance of a Multi-Key Access Method Based on Descriptors and Superimposed Coding Techniques", *Information Systems 10*, 4 (1985), 391-403.
23. R. Sacks-Davis, K. Ramamohanarao and A. J. Kent, "Multi-Key Access Methods Based on Superimposed Coding Techniques", Technical Report 87/4, Department of Computer Science, Royal Melbourne Institute of Technology (to appear ACM Trans. Database Systems, Dec. 1987), 1987.
24. S. Stiassny, "Mathematical Analysis of Various Superimposed Coding Schemes", *American Documentation 11*, 2 (February 1960), 155-169.
25. M. Wada, Y. Morita, H. Yamazaki, S. Yamashita, N. Miyazaki and H. Itoh, "A Superimposed Code Scheme for Deductive Databases", *ICOT Technical Report*, 1987.
26. M. J. Wise and D. Powers, "Indexing Prolog Clauses via Superimposed Code Words and Field Encoded Words", *1984 Int. Symp. Logic Programming*, Feb. 1984, 203-210.
27. H. K. T. Wong, H. Liu, F. Olken, D. Rotem and L. Wong, "Bit Transposed Files", *Proceedings of 11th Conference on Very Large DataBases*, Stockholm, August 21-23, 1985, 448-457.
28. G. Zipf, *Human Behaviour and the Principle of Least Effort: An Introduction to Human Ecology*, Hafner Publications, 1949.