Multikey Access Methods Based on Superimposed Coding Techniques

R. SACKS-DAVIS and A. KENT
Royal Melbourne Institute of Technology
and
K. RAMAMOHANARAO
University of Melbourne

Both single-level and two-level indexed descriptor schemes for multikey retrieval are presented and compared. The descriptors are formed using superimposed coding techniques and stored using a bit-inversion technique. A fast-batch insertion algorithm for which the cost of forming the bit-inverted file is less than one disk access per record is presented. For large data files, it is shown that the two-level implementation is generally more efficient for queries with a small number of matching records. For queries that specify two or more values, there is a potential problem with the two-level implementation in that costs may accrue when blocks of records match the query but individual records within these blocks do not. One approach to overcoming this problem is to set bits in the descriptors based on pairs of indexed terms. This approach is presented and analyzed.

Categories and Subject Descriptors: H.2.2 [**Database Management**]: Physical Design—*access methods*; H.3.2 [**Information Storage and Retrieval**]: Information Storage—*file organization*

General Terms: Design, Performance

Additional Key Words and Phrases: Descriptors, hashing, partial match retrieval, record signatures, superimposed coding

1. INTRODUCTION

Existing database systems generally fall into two classes. In the first class, the database systems are designed for formatted records and include relational database systems, while the second class contains systems designed for the retrieval of free text. There have been some attempts to design systems to handle both formatted and unformatted data [2, 9, 15, 19, 22]. These systems are particularly useful in the areas of office automation, computerized libraries, and image databases.

An essential component of these extended systems is an access method that can efficiently store and retrieve both formatted and unformatted data. One

Authors' addresses: R. Sacks-Davis and A. Kent, Department of Computer Science, Royal Melbourne Institute of Technology, 124 LaTrobe St., Melbourne, Victoria, Australia 3000; K. Ramamohanarao, Department of Computer Science, University of Melbourne, Parkville, Victoria, Australia 3052.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1987 ACM 0362-5915/87/1200-0655 $01.50

approach that has been advocated for these applications uses superimposed coding techniques to form record descriptors for the records in the data file [23, 24]. This method gives good retrieval performance and is efficient of storage [10, 21].

As well as forming descriptors for individual records in the file, it is possible to form descriptors for blocks of records, thus forming a multilevel descriptor file. For large databases, the multilevel index is more efficient. However, block descriptors can lead to the occurrence of *unsuccessful block matches*, which occur when a block of records contains the terms specified in a query but the individual records within these blocks do not.

In this paper we propose a two-level implementation of a superimposed coding scheme. The encoding scheme used to form the descriptors at the block level is designed to reduce the occurrences of unsuccessful block matches. The benefit of this method is that query times remain low for both single-term and multiterm queries and the storage overheads required are small. The method remains efficient, even for large data files containing hundreds of thousands of records, each record containing many terms.

This paper is organized as follows. First we present a brief overview of some of the methods used for text retrieval and describe some of the approaches, based on superimposed coding, that have been proposed. A description of our method is then presented in Section 3.

In Section 4 we compare the one-level and two-level implementations of the descriptor files. In order to compare these indexing schemes, a number of issues must be considered. These include the size of the file, the file system used on the host computer, whether dedicated hardware is available, and the query types that need to be supported. We describe the differences between the one- and two-level implementations of superimposed coding in a number of environments and show that for large data files the two-level implementation is generally more efficient than the one-level scheme for queries for which the number of matching records is small (0–200 records). In an interactive environment, such queries are the most common and must be supported very efficiently.

In Section 5, the encoding scheme, which takes into account the frequencies of the index terms, is presented and analyzed. We show that with this encoding scheme, the two-level scheme is efficient for single- and multiterm queries. Results computed from a theoretical analysis are presented in Section 6, and experimental results obtained from a library database containing 150,000 records are given in Section 7.

In Section 8 some practical issues are considered. The encoding technique that is proposed requires the identification of those indexed terms that appear in a large number of records (common words). We describe practical techniques for identifying these common terms.

In Section 9 a fast batch insertion algorithm is described. The proposed two-level scheme uses a storage technique called a bit slice implementation. This technique is required for efficient query processing but makes interactive insertions relatively slow. For applications involving large databases, fast insertion capabilities are extremely important and with the method proposed in this paper, batch insertion costs of 2 or 3 disk accesses per record can be achieved.

The conclusions are presented in Section 10.

2. OVERVIEW

Faloutsos [10] has presented a recent review of access methods applicable to both formatted and unformatted data. He classifies text retrieval methods into the following four classes:

(1) *Full Text Scanning.* In this method the full text database is scanned for matching records [1, 4, 17]. No extra storage overheads are incurred, but the method is relatively slow for large databases.

(2) *Inversion.* This method uses an inverted file index. It has been implemented in many commercial text retrieval systems [27, 28]. It provides relatively efficient query speeds, but can be very expensive of storage. In addition insertion times are slow.

(3) *Clustering.* In this method, similar documents are grouped together to form clusters [26, 27, 30]. Clustered documents can be stored physically together, facilitating efficient retrieval of related documents. A descriptor is stored for each cluster and a search correlates, typically using a vector similarity function, these descriptors with the query descriptor to retrieve relevant documents. The main disadvantage of these methods is the slow insertion times [10].

(4) *Signature Files.* In this method, a descriptor or signature is associated with each record or document [10], the descriptor being a bit encoding of the values used to retrieve the record. When a query is processed, the file of descriptors, rather than the data records, is examined for possible matches. A query descriptor is formed using the same encoding technique that is used for forming record descriptors. The possible record matches are those records whose descriptors contain bits set in each position for which a bit is set in the query descriptor. Signature file methods have good retrieval properties and require small storage overheads.

We propose a new signature file method suitable for large data files. We begin by briefly reviewing some signature file methods.

For methods to be applicable to both text retrieval as well as formatted data, an effective way to form the descriptors uses superimposed coding techniques [10]. With this method, descriptors are formed for each of the terms used to retrieve the record, and the record descriptor is formed in turn by superimposing (inclusive ORing) the term descriptors. A term descriptor is a bit vector of b bits with exactly k bits set to 1. The superimposing of term descriptors makes the method readily amenable to unformatted records, since there are no restrictions (other than for performance considerations) on the number of terms per record.

Various compression techniques have been proposed for representing the record descriptors. If b is large and k is small, then the descriptor will be sparse and compression techniques can be applied. McIlroy [18] proposed a compression technique for which the number of zeros between two successive 1s in the sparse vector was recorded. Faloutsos [11] proposed another compression technique for which the sparse vector is divided into groups of consecutive bits and variable length encodings of the bits set in each of these groups are stored. A survey of these compression-based representations is presented in [11]. Although efficient

of storage, these techniques make query processing slower and are not considered in this paper.

There have also been different techniques proposed for the storage of record descriptors. One approach, which we call a bit-slice representation, improves query response time by reducing the number of bits that have to be retrieved from the file of record descriptors at query time. This storage technique has been used in [20, 24, 32].

The signature file methods surveyed in [11] are all one-level schemes in the sense that descriptors are formed for single records only. As a consequence, these methods become relatively slow for large data files, since all of these descriptors have to be examined on query. It is possible to overcome this problem by forming descriptors for blocks of records and thereby implement a multilevel indexing scheme. A two-level scheme for superimposed coding was proposed in [24], the performance of which is analyzed in [25]. The idea of a multilevel indexed descriptor method was first proposed in [20]; however, the approach in that paper was based on disjoint coding techniques rather than superimposed coding, the latter being more efficient for text retrieval applications.

As well as using a two-level descriptor file to improve query response times, a bit slice representation is used for the storage of the block descriptors in the method proposed in this paper. A property of this representation is that individual bits belonging to particular block descriptors can be stored very far apart on the secondary storage device. As a consequence, although query times are reduced with this storage technique, insertion costs are no longer cheap. If insertions are batched, however, considerable savings can be made. In this paper we describe a batch insertion technique for which the cost of forming the block descriptor file using the bit slice representation is typically less than one disk access per record.

In order that the two-level scheme be efficient for all query types, it is important that the encoding scheme take into account the frequencies of the index terms. This is the approach adopted in this paper. The main purpose of the encoding scheme is to reduce the number of occurrences of unsuccessful block matches. Although unsuccessful block matches do not occur with one-level implementations, improved performance can also be achieved with these one-level schemes if the encoding scheme takes into account term frequencies. Encoding schemes for the one-level implementations that take into account the nonuniform distribution of the index terms were proposed in [20] and analyzed in [12].

The coding techniques used to form term descriptors can be extended so that direct indexing is possible on word parts and pairs of words. This makes the superimposed coding methods extremely attractive in a number of different applications. Harrison [13] proposed using superimposed coding for substring testing, and this approach is further explored in [11]. In this paper we describe some experimental results obtained when superimposed coding is used to directly index on word phrases.

In addition to text retrieval, superimposed coding techniques have been applied to a number of other application domains. These include message files [6], optical disk storage [7], statistical databases [32], filtering methods [3, 18], and Prolog databases [8, 22]. Superimposed coding techniques are well suited to hardware implementation [14, 16].

In this example, each record describes a book and has up to five terms ($s = 5$) corresponding to the three attributes: author name, publication date, and title keyword.

$$R = \begin{cases} v_1 = \text{Theroux, P.} & \text{(author name)} \\ v_2 = 1975 & \text{(publication date)} \\ v_3 = \text{Great} & \text{(title keyword)} \\ v_4 = \text{Railway} & \text{(title keyword)} \\ v_5 = \text{Bazaar} & \text{(title keyword)} \end{cases}$$

There will be three functions H_1, H_2, H_3 for generating codewords, one for each attribute. Suppose that $b = 15$ and $k = 2$ and

$$H_1(v_1) = 00010\ 00000\ 10000$$
$$H_2(v_2) = 01001\ 00000\ 00000$$
$$H_3(v_3) = 00010\ 00100\ 00000$$
$$H_3(v_4) = 00000\ 00010\ 00001$$
$$H_3(v_5) = 00001\ 00001\ 00000$$

Then D_R, the record descriptor for record R, is

$$D_R = 01011\ 00111\ 10001$$

Fig. 1. An example of superimposed coding.

3. DESCRIPTION OF THE METHOD

We begin by describing the one-level scheme. A record descriptor is a bit string that is formed from the terms that are used to retrieve the record. A record descriptor is constructed as follows. First each term is transformed into a code word that is a bit string of length b containing exactly k 1s and $b - k$ 0s. Here b and k are parameters of the method. Suppose that each record contains s terms. Then a record descriptor is formed by superimposing (inclusive ORing) the corresponding s code words for record R. An example of this process is given in Figure 1.

A query consists of the specification of q terms ($q \leq s$). These q values are transformed into q code words that are then superimposed to form a query descriptor. Observe that if a record satisfies a query, then every bit position that is set in the query descriptor must also be set in the record descriptor. Thus to answer a query it is necessary to search for record descriptors that match the query descriptor in this way. This can be achieved using simple AND and OR operations.

It is possible that a record descriptor matches a query descriptor but the corresponding record does not satisfy the query. Such an occurrence is referred to as a *false match*. The probability of a false match can be made arbitrarily small by appropriate choice of the parameters b and k.

As we have described the method so far, it is necessary to retrieve every record descriptor in order to answer a query. We now describe the two techniques discussed previously for reducing the amount of descriptor file that must be retrieved on query.

The first technique related to how the record descriptors are stored on disk. Suppose the data file contains N records. Rather than viewing the descriptor file as consisting of N strings of b bits in length, it is possible to store the file as b

strings, each of length N. If a query contains w 1s, then with the latter representation it is possible to check the relevant w bits of every record descriptor without fetching any of the other bits. Thus only wN rather than bN bits of the descriptor file need be examined on a query. This approach, known as the *bit slice* representation of the descriptor file, can contribute considerable savings, since typically $w \ll b$.

Even with the bit slice representation, a large amount of data must be fetched from secondary store in order to answer a query. Consider a query for which a single term is specified. Then k slices, each containing N bits, must be fetched from disk. In a shared environment, where the disk is organized as a number of equal sized pages or blocks, a large number of disk accesses (seeks) are required to retrieve these data. If each page in secondary store has a capacity of P bits, then the number of disk access required is at least $k \cdot \lceil N/P \rceil$. If $N = 500{,}000$ and $P = 8192$ (1K bytes), then for $k = 4$, as many as 248 disk accesses are required to determine the matching records. Even with a dedicated disk drive, the large amount of data to be processed makes the one-level scheme somewhat expensive.

Another problem with using a one-level scheme together with a bit slice representation is that a separate data structure mapping a logical record number onto a physical location on disk must be maintained. (This is not a problem if, for example, the data records are fixed length and the physical records are stored contiguously on disk.) If this data structure is stored on disk, then the cost of answering a query increases due to the indirection introduced.

The second technique for reducing the amount descriptor file that must be examined on query is based on using a *multilevel descriptor file* rather than a single-level file. In [24], a two-level scheme is proposed for which a data file of N records is viewed as consisting of N_s blocks, each containing N_r records where $N = N_s \cdot N_r$. Both block descriptors and record descriptors are stored in this scheme. A block descriptor is formed analogously to a record descriptor using all the terms of all the records contained in that block. It will, in general, be much larger than a record descriptor. Suppose that the block descriptors are characterized by parameters b_s and k_s and the record descriptors by b_r and k_r.

In the approach we propose (see Figure 2), the block descriptor file is stored using the bit slice representation, whereas the record descriptors are stored as bit strings. The physical location of a record is stored together with its corresponding record descriptor. We also assume that the parameters of the two-level scheme are such that all of the record descriptors for a particular block, together with their associated pointers, can fit entirely within a single page of secondary store.

In order to answer a query using the two-level scheme, two query descriptors are formed: a query block descriptor and a query record descriptor. The query block descriptor is based on parameters b_s and k_s and is used to determine which blocks satisfy the query. Only the record descriptors from matching blocks are then compared to the query record descriptor.

The cost of answering a query that specifies a single term can be estimated as follows. First the query block descriptor is formed by determining which of the b_s descriptor bits are set by this term. When only one term is specified, then

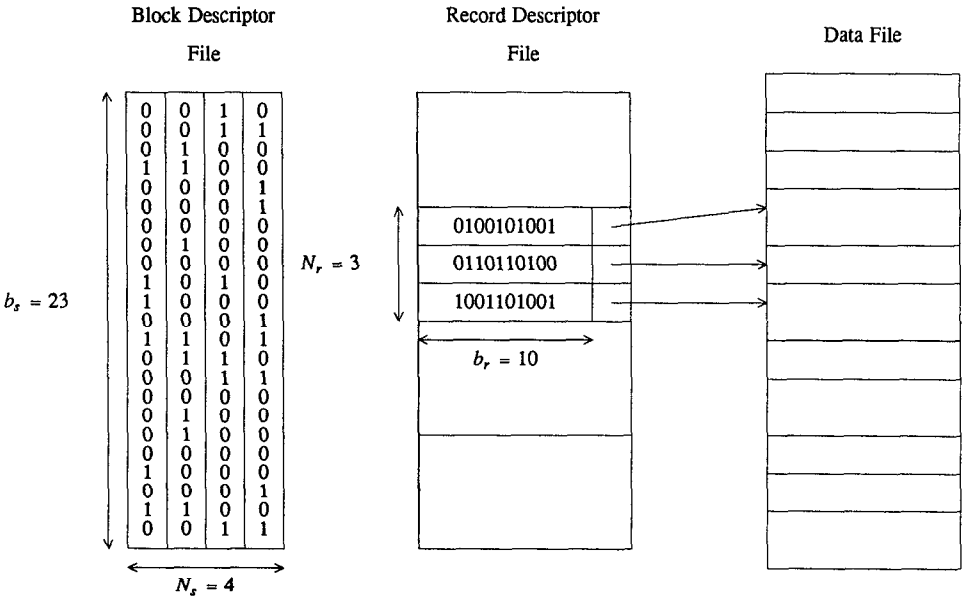

Fig. 2. Index structure for the two-level scheme.

exactly k_s bits are set. The corresponding slices of the block descriptor file are then fetched from secondary store. This will typically involve k_s disk accesses or a small multiple thereof, since the length of a slice is now N_s rather than N bits as in the single-level scheme. It is then possible to determine which blocks satisfy the query. For each matching block, we fetch the corresponding record descriptors. By design, all the record descriptors for a given block fit within a single page so that if there are n_s matching blocks, this can be achieved with n_s further disk accesses. It is then possible to determine which records satisfy the query and fetch the corresponding data records. Suppose that A records satisfy the query and $D(A)$ disk accesses are required to fetch all these records from disk. The total number of disk accesses, therefore, required to answer a query for which a single term is supplied is $k_s + n_s + D(A)$. Here we have ignored false matches, but typically the number of false matches will be small by design.

For queries that specify a single term, the two-level scheme overcomes both of the problems previously discussed for the one-level implementation. Let us now consider queries for which more than one term is specified. In this case $w_s \geq k_s$ bits will be set in the query descriptor and up to w_s slices from the block descriptor file can be retrieved in order to determine the number of matching blocks. Note, however, that only k_s slices need be fetched in order that the probability of a false match with a block containing none of the query terms be the same as for the single-value query case. As a consequence, the number of disk accesses required to determine the matching blocks for multiterm queries is typically only marginally greater than the constant overhead required for single-term queries.

For queries that specify two or more terms, an additional cost, not associated with the single-level scheme, occurs in the two-level implementation. Consider a

query that specifies two terms, a_1 and a_2. Because a block descriptor is formed from all of the terms of the records belonging to that block, there may exist a block that contains a record with term a_1 and contains another record with term a_2 but contains no record with both terms a_1 and a_2. Such a block will generate a block match even though it contains no records that satisfy the query. We refer to this occurrence as an *unsuccessful block match.* The cost of an unsuccessful block match is one disk access, since the page of record descriptors for that block will be fetched. It is very unlikely that any false record matches will be generated as a result of an unsuccessful block match [25]. This is because many more than k_r bits will be set in the query record descriptor. As the record descriptors are stored as bit strings, all the set bits in the query descriptor will be compared to the corresponding bits in each record descriptor that is examined. Hence the probability of a false record match is much lower than for the single-term case.

In order to estimate the potential number of unsuccessful block matches, let A_j be the number of records containing term a_j, $j = 1, 2$, and let A^* be the number of records containing both a_1 and a_2. Define

$$p(A_j) = 1 - \left\{1 - \frac{A_j}{N}\right\}^{N_r}.$$

Then $p(A_j)$ is an estimate of the probability that a particular block contains one or more records with term a_j. This estimate is based on the assumption that records with term a_j are distributed uniformly over the data blocks. If the terms a_1 and a_2 are independent, then we may estimate the number of unsuccessful block matches, $U(a_1, a_2)$ as

$$U(a_1, a_2) \approx N_s \cdot P(A_1 - A^*) \cdot P(A_2 - A^*) \cdot [1 - P(A^*)].$$

For $A_j \ll N$, we may approximate $P(A_j)$ by

$$P(A_j) \approx \frac{A_j N_r}{N} = \frac{A_j}{N_s}.$$

If we consider the case $A_1 = A_2 = A$, $A^* = 0$, then with this approximation

$$U(a_1, a_2) \approx \frac{A^2}{N_s}.$$

We can therefore determine the number of occurrences, x, of two such terms, necessary to result in a single unsuccessful match as

$$x = (N_s)^{1/2}.$$

If $A = \omega x$, then the number of unsuccessful matches can be estimated as ω^2.

When $n > 2$ terms are supplied in a query, we can estimate the number of unsuccessful block matches $U(a_1, a_2, \ldots, a_n)$ in a similar way to the case $n = 2$. In particular, if $A_1 = A_2 = \cdots = A_n = A$, and if $A^* = 0$, then

$$U(a_1, a_2, \ldots, a_n) \approx \frac{A^n}{(N_s)^{n-1}}.$$

Obviously the numbers of unsuccessful block matches will tend to decrease as more terms are supplied in a query.

One way to substantially reduce unsuccessful block matches is to set bits in the block descriptors by hashing *pairs of terms* as well as the terms themselves. Thus if there exist two terms a_1 and a_2 in a record, then it is possible to set bits in the appropriate block descriptor based on the pair $a_1 \cdot a_2$ in addition to setting bits generated from a_1 and a_2. To answer a query specifying both a_1 and a_2, these extra bits will be set in the query descriptor at the block level, thereby eliminating most of the unsuccessful block matches. This approach will be particularly useful if the attributes corresponding to both a_1 and a_2 have domains that are small compared to the number of records in the file or if the attributes have highly nonuniform distributions. A method employing this technique is described in Section 5 of this paper. The extra overheads generated by setting these extra bits are small, since typically only one or two bits need be set for a pair of terms and this technique need only be applied to selected terms.

We conclude this section by remarking that the generation of bits in a descriptor using word pairs has other applications. By setting a single bit for every pair of *adjacent* words in a string of text, it is possible to directly support, at small cost, the indexing of *text phrases* as well as single words of text. When a phrase is specified in a query, the use of such adjacency bits will practically eliminate the retrieval of those documents that contain all the words in the phrase, unless these words are in consecutive locations. Another application to text retrieval arises when triplets of consecutive letters rather than single words are used for indexing [12]. This approach is useful for supporting queries in which substrings of words are specified. Unsuccessful matches can occur if this technique is adopted, however, since records that contain all the triplets specified in a substring will be retrieved, even when these triplets are in the incorrect order. Setting bits based on pairs of consecutive triplets can be used to reduce the consequent cost.

4. COMPARISON OF THE ONE-LEVEL AND TWO-LEVEL SCHEME

We begin by investigating the costs when a single term is supplied in a query. In order to estimate the number of false matches, we will use the formula

$$q(t; k, b, s) = \left\{1 - \left(1 - \frac{k}{b}\right)^s\right\}^t$$

as an estimate of the probability that t chosen bits are set in a descriptor that is defined by parameters b and k and which is formed from s terms. If the number of descriptors that must be examined when answering a query is n and the number of bits set in the query descriptor is t, then we can estimate the number of false matches as $(n - A) \cdot q(t; k, b, s)$, where A is the number of true matches. Since $A \ll n$, we approximate this estimate by $n \cdot q(t; k, b, s)$ to simplify the analysis.

In the following we present the costs of both the one- and two-level schemes. As discussed previously, one of the components in the cost is the time required to retrieve a bit slice from secondary store. For the one-level scheme, these slices will be N bits in length, where N is the number of records to be indexed. In a shared system where files are stored in noncontiguous blocks (pages), several seeks will typically be required to retrieve a single slice from secondary memory.

In this environment the one-level scheme can become very expensive compared to the two-level scheme. On the other hand, in a dedicated environment where the descriptor file is stored contiguously on disk and hardware is provided to enable the retrieval of a single slice in only one disk access, then the one-level scheme becomes more competitive. We will compare the one-level and two-level schemes in both of these environments. For definiteness, we will refer to the former system environment as being shared and the latter as being dedicated.

An assumption that we will make is that the data records are of variable length. For the one-level scheme, a separate array of pointers to the data records must be maintained. For large files this array will often be required to be resident on disk. We will, however, also investigate the case when the array is stored in core.

Finally, we also assume that there is sufficient in core memory available to store two bit slices.

In the comparison, we make use of the following parameters.

P	Page size in bits
w	Word size = pointer size
B	Blocking factor (number of data records per block)
T_s	Average seek time
T_d	Time required to read one bit of data from disk (inverse of data transfer rate)
T_a	Time required for an AND operation on a pair of bits
T_z	Time required to detect a zero word/word size
T_o	Time required to locate the position of a 1 within a word
T_c	Average computation time per block to retrieve the matching records
A	Number of records that satisfy the query
C_i	Cost of step i for the one-level scheme
D_i	Cost of step i for the two-level scheme

In the following, we will require an estimate of the expected number of blocks containing r randomly chosen tokens, given that m is the number of tokens per block and n is the number of blocks. An estimate of the probability that an arbitrary block contains one or more of the tokens, $e(r, m, n)$, is given by the following formula [33]:

$$e(r, m, n) = \left\{1 - \left\{1 - \frac{m}{mn}\right\}\left\{1 - \frac{m}{(mn-1)}\right\} \cdots \left\{1 - \frac{m}{(mn-r+1)}\right\}\right\}.$$

The expected number of blocks containing the r tokens is then $e(r, m, n) \cdot n$. In order to reduce the computation time required to evaluate $e(r, m, n)$, a closed, noniterative approximation was provided by Whang et al. [31]. The noniterative formula is

$$e(r, m, n) \approx \left[1 - \left(1 - \frac{1}{n}\right)^r\right] + \left[\frac{1}{n^2 m} \times \frac{r(r-1)}{2} \times \left(1 - \frac{1}{n}\right)^{r-1}\right] + \left[\frac{1.5}{n^3 m^4} \times \frac{r(r-1)(2r-1)}{6} \times \left(1 - \frac{1}{n}\right)^{r-1}\right]$$

when $r \leq mn - m$ and

$$e(r, m, n) = 1$$

when $r > mn - m$. This approximation was used throughout this paper.

It should be pointed out that the calculation of expected block accesses using the average number of tokens per block, m, rather than the distribution can lead to overestimates of the block accesses required if there is a variable number of tokens per block [5]. In the following comparison, the costs at the index levels will be estimated accurately, since the descriptors are of fixed length. We will also assume that the data records have small variance in length. If this is not true, then the estimates for both methods will be affected, since the estimated costs of retrieving the actual data from disk will be biased.

We begin by considering the cost of answering a query using the one-level scheme. If we assume that a single term has been supplied in the query, then the one-level scheme consists of the following steps.

Algorithm 1 (*one-level*)

Step 1. Retrieve and transfer k bit slices from disk. If the scheme is implemented using a dedicated disk, then the cost of this step is

$$C_1 = k \cdot T_s + k \cdot N \cdot T_d.$$

On the other hand, if the secondary memory is shared and a seek is required for every page that is retrieved from secondary store, then the cost of step 1 becomes

$$C_1 = k \cdot \left\lceil \frac{N}{P} \right\rceil \cdot (T_s + P \cdot T_d).$$

Step 2. AND k bit slices

$$C_2 = (k - 1) \cdot N \cdot T_a.$$

Step 3. Find the positions of the set bits in the resultant vector

$$C_3 = N \cdot T_z + n \cdot T_o,$$

where

$$n = \text{true record matches} + \text{false record matches} = A + F$$

and

$$F = N \cdot q(k; k, b, s).$$

Step 4. Determine the locations of the matching n records. If the array of pointers to the data records is stored on disk, then the cost of this step is

$$C_4 = e\left(n, \frac{P}{w}, \frac{Nw}{P}\right) \cdot [T_s + P \cdot T_d].$$

If the array is kept in core, then the cost of step 4 can be ignored.

Step 5. Retrieve data

$$C_5 = e\left(n, B, \frac{N}{B}\right) \cdot [T_s + P \cdot T_d + T_c].$$

We now present the costs of the two-level scheme. The steps involved are described below.

Algorithm 2 (*two-level*)

Step 1. Retrieve and transfer k_s bit slices from the block descriptor file stored on disk. In a dedicated environment the cost is

$$D_1 = k_s \cdot T_s + k_s \cdot N_s \cdot T_d,$$

while in a shared environment, the cost is

$$D_1 = k_s \cdot \left\lceil \frac{N_s}{P} \right\rceil \cdot (T_s + P \cdot T_d).$$

Step 2. AND k_s bit slices

$$D_2 = (k_s - 1) \cdot N_s \cdot T_a.$$

Step 3. Find the positions of the set bits in the resultant vector

$$D_3 = N_s \cdot T_z + n_s \cdot T_o,$$

where

$$n_s = \text{true block matches} + \text{false block matches} = e(A, N_r, N_s) + F_s$$

and

$$F_s = N_s \cdot q(k_s; k_s, b_s, N_r s).$$

Step 4. Read record descriptors for matching blocks

$$D_4 = n_s \cdot [T_s + P \cdot T_d].$$

Step 5. Determine the matching records by examining the record descriptors

$$D_5 = n_s \cdot [(k_r - 1) \cdot N_r \cdot T_a + N_r \cdot T_z + c \cdot T_o],$$

where

$$c = \frac{n_r}{n_s} \approx 1$$

and

$$n_r = \text{true record matches} + \text{false record matches} = A + F_r$$

and

$$F_r = N_r \cdot n_s \cdot q(k_r; k_r, b_r, s).$$

For simplicity, we have assumed that the record descriptors are also stored in bit slice form in deriving D_5.

Step 6. Retrieve data

$$D_6 = e\left(n_r, B, \frac{N}{B}\right) \cdot [T_s + P \cdot T_d + T_c].$$

In order to compare the two schemes, let us assume initially that a dedicated disk drive is available. If we choose $k = k_s$ and we also assume that the number of matching records is small, so that $n_s \approx n \ll N$, then both schemes will require the same number of seeks to answer a query for which a single term is specified. The difference in costs for the two schemes reduces to the following:

$$\begin{aligned} DIFF &= \text{cost of one-level scheme} - \text{cost of two-level scheme} \\ &= (N - N_s)[k_s \cdot T_d + (k_s - 1) \cdot T_a + T_z] - n[(k_r - 1) \cdot N_r \cdot T_a + N_r \cdot T_z + T_o]. \end{aligned}$$

On most computers $T_a \approx T_z$, so the difference becomes

$$(N - N_s)[T_d + T_a]k_s - n[k_r \cdot N_r \cdot T_a + T_o].$$

Since the first term of the preceding expression dominates the second term for typical parameter values, we can conclude that $DIFF > 0$ under these assumptions. For queries that result in a small number of matching records, the one-level scheme is more expensive than the two-level scheme. On the other hand, as the number of matching records, A, increases, it will be seen that the one-level scheme becomes more competitive. The relationship between the two schemes for typical parameter values is explored further below.

A number of experiments were conducted, comparing the two schemes using databases of different sizes. For these tests, the parameters of the two-level scheme were chosen using the following criteria. The number of blocks was chosen to be an integral multiple of the page size, P, so that the bit slices were aligned on page boundaries. The number of records per block was chosen so that $N_r \cdot (b_r + w) = P/2^i$ for some integer $i \geq 0$. It could then be guaranteed that the record descriptors for a matching block could be retrieved using a single disk access. For each database, b_s was chosen so that the expected number of false block matches was one.

The parameters of the one-level scheme were chosen so that $k = k_s$ and the storage overhead generated by the one-level scheme was equal to that of the two-level scheme.

As an example, consider the following database of just over one million records, each containing 20 terms. The parameters used in the comparisons are listed in Figure 3.

The times in seconds to answer a query for various numbers of matching records are presented in Figure 4. It can be observed that if the number of matching records, A, is sufficiently large, then the one-level scheme is faster than the two-level scheme, but when A is small, the two-level scheme is more efficient. The crossover point depends on the system environment. In a shared environment, the two-level scheme is faster for queries with up to 2000 matching records. In a dedicated environment, the crossover point is approximately 500 records if, for the one-level scheme, the array pointers to the data records is stored on disk and approximately 90 records if it is stored in core.

N = 1,048,576 (number of records)
s = 20 (number of attributes)
P = 8192 (page size (1K bytes))
B = 8 (average record size 128 bytes)

T_s = 35 milliseconds
T_d = 0.2 microseconds (5 megabits/sec)
T_a = 0.2 microseconds
T_z = 0.2 microseconds
T_o = 0.2 microseconds
T_c = 500 microseconds

Two-Level Scheme
N_s = 40,960 (number of blocks)
N_r = 26 (records per block)

$k_s = 4$ $\quad k_r = 8$
$b_s = 28{,}540$ $\quad b_r = 283$
$q(k_s; k_s, b_s, s \cdot N_r) = 1/N_s$ $\quad q(k_r; k_r, b_r, s) = 1/29N_r$

One-Level Scheme
$k = 4$
$b = 1397$
$q(k; k, b, s) \approx 10/N$
Storage = 8.93 bytes per indexed term

Fig. 3. Example database.

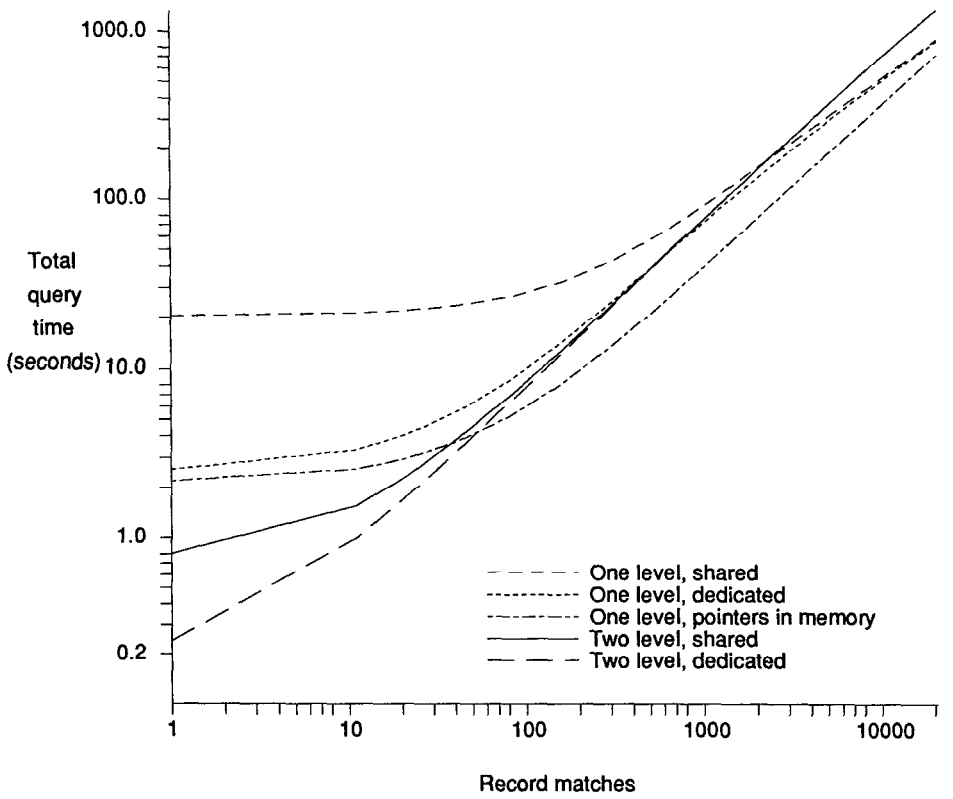

Fig. 4. Query costs for page size = 1024 bytes, s = 20, N = 1,048,576.

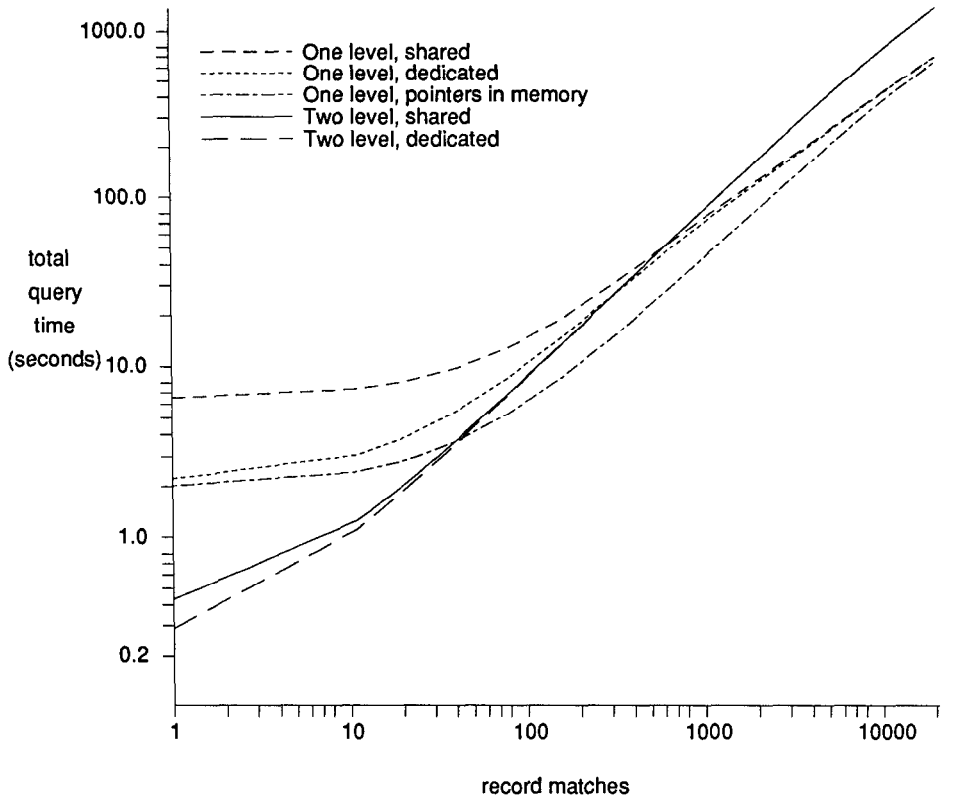

Fig. 5. Query costs for page size = 4096 bytes, $s = 10$, $N = 1{,}048{,}576$.

As the page size increases, and the database gets smaller, the values of A at the crossover points tend to decrease. In Figure 5, the page size is increased to 4K bytes, and s is reduced to 10. The values at the crossover points described above are 600, 250, and 40, respectively. For a page size of 1K bytes, a database of 131,072 records, and $s = 10$, the values are 300, 120, and 8, respectively.

If we consider now queries for which more than one term is supplied, then for the two-level scheme there is an additional cost involved due to unsuccessful block matches. This additional cost is not present in the one-level scheme. In order to estimate the effect of unsuccessful block matches on query performance, we computed for each value of A for which the two-level scheme outperformed the one-level scheme, the number of unsuccessful block matches, $u(A)$, that could be tolerated by the two-level scheme before becoming more expensive than the one-level implementation.

$$u(A) = \frac{\text{cost of one-level scheme} - \text{cost of two-level scheme}}{\text{cost of an unsuccessful block match}}.$$

The cost of an unsuccessful block match is the time required to retrieve and examine a block of matching record descriptors. This cost is $T_s + P \cdot T_d + q \cdot (k_r - 1) \cdot N_r \cdot T_a + N_r \cdot T_z$. Here we have assumed that no false record

	$N = 1{,}048{,}576$; $s = 20$; $P = 8192$			
A	Two-level shared	One-level shared	One-level dedicated	One-level dedicated/in-core
1280	0.075	0.084	0.070	0.039
5280	0.072	0.061	0.058	0.037
10,240	0.070	0.051	0.050	0.036
	$N = 1{,}048{,}576$; $s = 10$; $P = 32{,}768$			
A	Two-level shared	One-level shared	One-level dedicated	One-level dedicated/in-core
1280	0.083	0.070	0.067	0.043
5120	0.079	0.047	0.046	0.038
10,240	0.075	0.041	0.041	0.036
	$N = 131{,}072$; $s = 10$; $P = 8192$			
A	Two-level shared	One-level shared	One-level dedicated	One-level dedicated/in-core
1280	0.070	0.051	0.050	0.036
5120	0.060	0.037	0.036	0.032
10,240	0.050	0.030	0.030	0.028

Fig. 6. Cost, per record, of answering a query in seconds for large A.

matches will occur as a result of an unsuccessful block match. In the next section we provide an encoding scheme for which the number of unsuccessful matches is almost always between 0 and 1 per matching record. Thus for queries that supply more than one value, the two-level scheme will almost certainly be less expensive than the one-level scheme if $u(A) \geq A$. The point at which $u(A) = A$ represents a crossover value for multiterm queries in the sense that as the number of matching records increases above this value, we can no longer guarantee that the two-level scheme is more efficient.

For the database described in Figure 3, if we consider a shared environment, the crossover point for multiterm queries is approximately 500 matching records compared to the value of 2000 records for single-value queries. We can conclude that for a particular multiterm query, the crossover point will be some value between 500 and 2000 records. Using the same argument for multiterm queries in the dedicated environment, the crossover point is some value between 50 and 500, and if in addition the array of pointers to the data records is kept in core for the one-level scheme, the crossover value is between 20 and 90.

In Figure 6, the costs per matching record of answering a single-term query are given for various configurations when the number of matching records, A, is large. If we discuss these costs in terms of disk accesses, then for the two-level scheme, the cost per matching record is typically 2 disk accesses. It decreases slowly with increasing A. If we take unsuccessful block matches into account, then for queries that supply a number of terms, the cost can be as high as 3 disk access. For the one-level scheme, the cost decreases from 2 disk access to 1 disk access as A increases. If the array of pointers is not stored in core, then it makes little difference whether the environment is shared or dedicated. If, however, the

pointer array is kept in core, then the cost remains close to 1 disk access for all values of A.

In an interactive environment the performance of a method for queries for which the number of matching records is small is very important. In an information retrieval system, an interactive user would typically only be interested in viewing a small number of records in response to a query. If a query is given for which the number of matching records is large, the user is typically required to supply more search terms to narrow the query response. The results of the tests show that in a system where the files are stored in noncontiguous blocks, the two-level scheme is clearly superior for this type of application. Even in a dedicated environment where the descriptor files are stored contiguously on disk and for which single seeks are sufficient to retrieve very large volumes of data, the two-level scheme is faster for typical queries for files that are sufficiently large. If, however, sufficient main memory is available to store the array of pointers to the data records in core, then the one-level scheme becomes more competitive, for smaller files in particular. In order to determine when the array can be kept in core, a number of factors must be considered. For example, the main memory requirements will be substantial; a file of one million records will require about two megabytes of memory, unless some time tradeoffs are made. Also, in a multiuser system, it will be required that separate users share the same memory, and this is operating system dependent.

5. AN ENCODING SCHEME TO REDUCE UNSUCCESSFUL BLOCK MATCHES

In order to model the behavior of the two-level scheme when more than one term is supplied in a query, it is necessary to investigate the effect of unsuccessful block matches on query performance. The aim is to produce an encoding scheme for which the number of unsuccessful block matches is small. In this section we describe a simple scheme for reducing unsuccessful block matches and then propose a more robust scheme. A list of the parameters used in this section is given in Figure 7.

Unsuccessful block matches are present due to the occurrence of particular data terms in large numbers of records. In the following we assume that the number of terms (tokens) stored in the database is $N_V = N \cdot s$, while the number of *distinct* values is N_D. If we order the distinct values v_i, $i = 1, 2, \ldots, N_D$ by the frequency of their occurrence, then v_1 is the most commonly occurring data value, followed by v_2, and so on. We will assume that the probability of occurrence, p_i, of the ith value within the data file follows the well-known Zipfian distribution [34]:

$$p_i = \frac{\alpha}{i^\beta}$$

(Zipf's law) for parameters α and β.

Because the highest ranked values will appear in a large number of blocks, we will modify the coding scheme so that these values will be treated differently from the less frequently occurring data values. A data value will be referred to as

N	number of records in database;
s	average number of terms per record;
N_s	number of blocks;
N_r	number of records per block;
b_s	length of a block descriptor;
b_r	length of a record descriptor;
k_s	bits set by a term in a block descriptor;
k_r	bits set by a term in a record descriptor;
l_s	number of bits set by pairs of common words;
C	number of common words;
C_1, C_2, C_3	three classifications of common words;
Δ	cutoff for the number of block matches for combination bits;
f	maximum value of unsuccessful block matches per true block match within region covered by common words;
μ	average density of the block descriptor file (bits 1 through b_s);
μ_D	probability of a bit being set by a regular word in the block descriptor file (bits 1 through b_s);
μ_C	probability of a bit being set by a pair of common words in the block descriptor file (bits 1 through b_s);
v_i	value i, where i is the rank of the term;
p_i	probability of occurrence of v_i;
α, β	parameters of Zipf's distribution;
s_D	average number of distinct terms per record;
s_C	average number of common terms per record;
N_V	total number of terms in a database ($N \cdot s$);
N_D	number of distinct terms in a database;
rcd_i	probability of v_i appearing in a record;
blk_i	probability of v_i appearing in a block;
$rec_{i,j}$	probability of a record containing v_i and v_j;
$blk_{i,j}$	probability of a block containing a record with terms v_i and v_j;
$true_{i,j}$	number of blocks containing records with v_i and v_j;
$total_{i,j}$	number of blocks containing both v_i and v_j;
$ubm_{i,j}$	number of unsuccessful block matches resulting from a query containing v_i and v_j.

Fig. 7. Parameters for the two-level scheme.

a *common word* if its rank is between 1 and C, where C is a parameter of the coding scheme. Values with rank $C + 1$ to N_D will be referred to as *regular* words.

Regular words are treated exactly as described in the previous section. For each regular word appearing in a record, a code-word, characterized by parameters b_s and k_s, is formed and superimposed on to the appropriate block descriptor. Common words are treated differently. In [23], Roberts suggested that rather than setting k_s bits in a block descriptor for a commonly occurring value, it is better to set a smaller number of bits. In the coding scheme we propose, a single bit only will be set for a common word. Moreover, this bit will be set in a disjoint field that is allocated to each block descriptor for common words. For each common word, one additional slice is allocated to the block descriptor file, so that a block descriptor contains $b_s + C$ bits rather than b_s bits. A bit set in the ith position of slice $b_s + j$ will indicate that there is a record in the ith block containing v_j, the jth ranked common word.

In order to eliminate unsuccessful block matches, bits will also be set in the block descriptor file *using pairs of common words*. We will refer to those bits set

in the block descriptor file by pairs of words as *combination bits*. Suppose a record contains m common words. Then there are $\binom{m}{2}$ pairs of common words. For each pair, a code word of length b_s with exactly l_s bits set will be formed and superimposed onto the first b_s bits of the appropriate block descriptor. Thus bit positions 1 to b_s are reserved for regular words and pairs of common words. Note that with this method, a record containing m common words will set at most $(s - m) \cdot k_s + m + l_s \cdot \binom{m}{2}$ bits in a block descriptor.

At the record descriptor level, no distinction between common and regular words is made. The scheme used is unchanged from the previous section.

The encoding method assumes that all terms are indexed. Of course, it is possible to treat the very common words as noise words and set no bits for these words in the descriptor files. The problem with this approach is that although it is not likely that common words will be specified in single-term queries, a number of common words may be specified together in a query that results in a small number of matching records. Ignoring common words entirely means that the latter queries cannot be answered efficiently.

First we show how to estimate the bit density of the block descriptor file when this encoding scheme is used. This estimate is required to predict the performance of the method. Given that the probability of occurrence of the ith value, v_i, is p_i, we can estimate the probability that v_i appears in a record as rcd_i where

$$\text{rcd}_i = e(p_i \cdot N \cdot s, s, N).$$

The number of distinct values per record, s_D, is then

$$s_D = \sum_{i=1}^{N_D} \text{rcd}_i \leq s.$$

Similarly, the average number of common words per record, s_C, can be estimated as

$$s_C = \sum_{i=1}^{C} \text{rcd}_i.$$

We can estimate the probability of v_i appearing in a block as blk_i, where

$$\text{blk}_i = e(N \cdot \text{rcd}_i, N_r, N_s).$$

We need to determine the probability that a bit in position 1 to b_s of a block descriptor is set. The probability that it is set by a regular word, μ_D, can be estimated as

$$\mu_D = 1 - \prod_{i=C+1}^{N_D} \left(1 - \frac{k_s}{b_s} \text{blk}_i\right).$$

We now consider pairs of common words. Let $\text{rcd}_{i,j}$ be the probability that both the ith and jth ranked values appear together in a record. The probability, μ_C, that an arbitrary bit is set by a pair of common words is

$$\mu_C = 1 - \prod_{i=1}^{C-1} \prod_{j=i+1}^{C} \left(1 - \frac{l_s}{b_s} \text{blk}_{i,j}\right),$$

where

$$\text{blk}_{i,j} = e(N \cdot \text{rcd}_{i,j}, N_r, N_s).$$

If we assume that the data values are independent, then we can estimate $\text{rcd}_{i,j}$ by $\text{rcd}_i \cdot \text{rcd}_j$.

The average density of the block descriptor file (here we are only concerned with bits 1 through b_s) is then μ where

$$\mu = 1 - (1 - \mu_D)(1 - \mu_C).$$

The expected number of false block matches when a single regular word is supplied in a query is $\mu^{k_s} \cdot N_s$. If a single common word is supplied, then no false block matches occur with the coding scheme used. In order to evaluate the performance of the method when two values are supplied, we have to consider the number of unsuccessful block matches that can occur. The effect of the coding scheme for queries for which two common words are specified, is that the number of unsuccessful block matches will reduce by a factor of μ^{l_s} compared to the previous scheme. This is because when answering such a query, the bit slices corresponding to that pair of common words are retrieved. For queries that contain a pair of attribute values that includes at least one regular word, the number of unsuccessful block matches is not so reduced.

Suppose the rank of the two values supplied are i and j, respectively, with $i < j$. The number of blocks containing records with both values is $\text{true}_{i,j}$, where

$$\text{true}_{i,j} = \text{blk}_{i,j} \cdot N_s = e(N \cdot \text{rcd}_i \cdot \text{rcd}_j, N_r, N_s) \cdot N_s.$$

The number of blocks containing both values v_i and v_j is $\text{total}_{i,j}$, where

$$\text{total}_{i,j} = N_s \cdot \text{blk}_i \cdot \text{blk}_j.$$

The number of unsuccessful block matches resulting from a query which specifies two values is then

$$\text{ubm}_{i,j} = \begin{cases} \mu^{l_s}(\text{total}_{i,j} - \text{true}_{i,j}), & \text{if } i,j \le C \\ \text{total}_{i,j} - \text{true}_{i,j}, & \text{otherwise.} \end{cases}$$

The number of unsuccessful block matches relative to the number of true matches is $\text{ubm}_{i,j}/\text{true}_{i,j}$. This value determines the additional cost in disk accesses per matching record of answering a query that is attributable to unsuccessful block matches.

If we make the first-order approximation $\text{blk}_i \approx N_r \cdot \text{rcd}_i$, then

$$\begin{aligned} \text{total}_{i,j} &\approx N_s \cdot N_r^2 \cdot \text{rcd}_i \cdot \text{rcd}_j, \\ \text{true}_{i,j} &\approx N_s \cdot N_r \cdot \text{rcd}_i \cdot \text{rcd}_j, \\ \text{total}_{i,j} - \text{true}_{i,j} &\approx N_s \cdot N_r \cdot (N_r - 1) \cdot \text{rcd}_i \cdot \text{rcd}_j, \end{aligned}$$

and

$$\frac{\text{ubm}_{i,j}}{\text{true}_{i,j}} \approx \begin{cases} \mu^{l_s}(N_r - 1), & \text{if } i,j \le C \\ N_r - 1, & \text{otherwise.} \end{cases}$$

This suggests the following design rule for computing the block descriptor parameters. In order that the number of unsuccessful block matches per matching record be limited to one, say, C should be chosen sufficiently large and μ chosen so that

$$\mu^{l_s} \cdot (N_r - 1) = 1.$$

Also, in order that the number of false block matches per query that occur when a single attribute-value is supplied is limited to say, one, we have a second constraint on μ, namely

$$\mu^{k_s} \cdot N_s = 1.$$

Combining these two criteria, we have

$$\mu = \min\left[\left(\frac{1}{N_s}\right)^{1/k_s}, \left(\frac{1}{N_r - 1}\right)^{1/l_s}\right].$$

As an example, consider a database for which N = 150,000 and s = 10. The number of blocks, N_s = 15,000 and the number of records per block N_r = 10. Suppose the data are Zipf distributed, with α = 0.08137 and β = 1.0. (With these values, the frequency of the data value of lowest rank, v_{N_D}, will be 1, and the value of β is typical of the values that are observed in much naturally occurring data [34].) In Figure 8, the values of $\text{ubm}_{i,j}/\text{true}_{i,j}$ are plotted for various values of i and j. It can be observed that as i and j increase, the ratio approaches N_{r-1} as predicted. These values were computed assuming the number of common words C = 0.

We have not discussed how many common words are typically required to make effective use of the preceding encoding scheme. In order to determine how many common words are necessary, a number of factors should be considered. Although the ratio $\text{ubm}_{i,j}/\text{true}_{i,j}$ grows with increasing i and j, the number of block matches decreases. We would like to be able to set combination bits for those queries for which the number of block matches exceeds some cutoff value, say Δ. Let us consider a query that specifies two values v_i and v_j as a point (i, j) in two-dimensional space, and draw a curve connecting those values of i and j for which the total block matches, $\text{total}_{i,j}$, equals a given value, Δ. A series of such curves are presented for the example database in Figure 9. If we consider all possible queries that supply two values, then for the given cutoff value, Δ, the area that we would like to cover lies between the corresponding curve and the axes, since this area contains the points (i, j) corresponding to queries for which the total block matches exceed Δ. The area to the right of the curve contains the pairs (i, j) corresponding to inexpensive queries.

The scheme we have just described, which is based on a set of C common words, covers queries for which the ranks of the two values are less than or equal to C (the square shaped area of Figure 10). In order to cover all points for which $\text{total}_{i,j} \leq \Delta$, a very large number of common words will be required, even for quite high values of Δ. The scheme becomes impractical to implement, since all of these common words must be stored in memory and the overheads become too large.

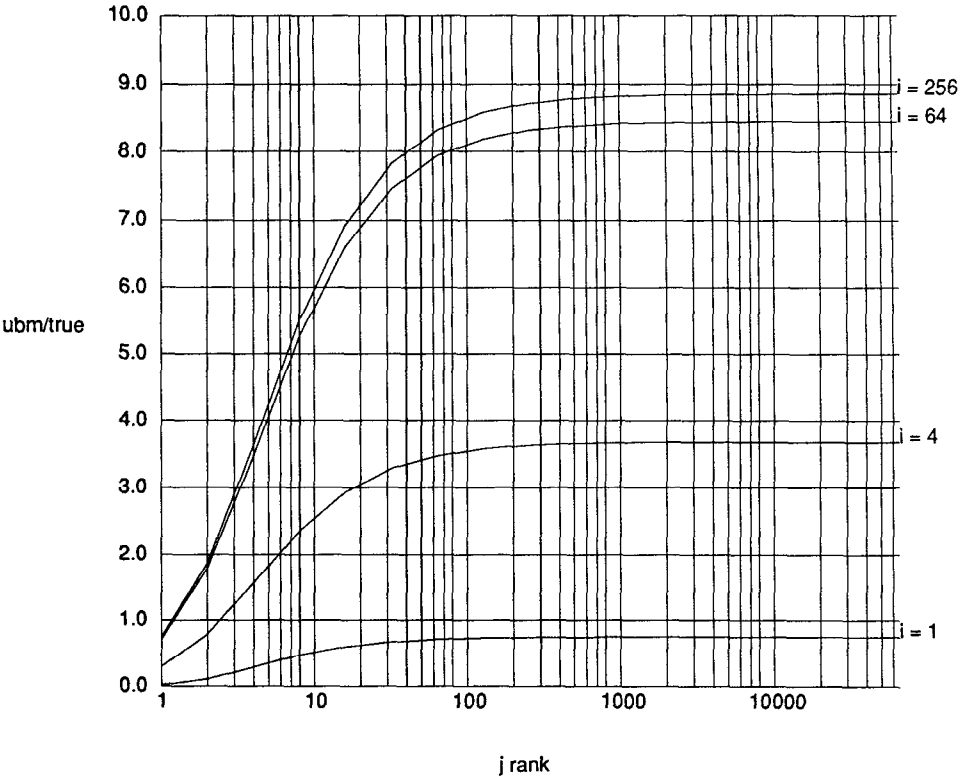

Fig. 8. $ubm_{ij}/true_{ij}$ vs. j for $i = 1, 4, 64, 256$.

To better approximate the desired area to be covered, we now consider a coding scheme based on three parameters C_i, $i = 1, 2, 3$, where $1 \leq C_1 \leq C_2 \leq C_3 \leq N_D$. This scheme identifies three sets of common words, and we will refer to a word whose rank lies between 1 and C_i as a C_i-Common word, $i = 1, 2, 3$. Words with rank greater than C_3 will be referred to as regular words. With these parameters, combination bits will be set for values (i, j) within the shaded area of Figure 11. It will be seen that unlike the previous scheme, the new scheme is easy to implement. Only C_2-Common words will need to be stored explicitly. The C_3-Common words that have rank greater than C_2 can be determined using an inexpensive filtering technique and need not be stored explicitly. This is important, since in practice C_2 will be relatively small and C_3 will be very much greater than C_2.

When inserting a record into the database, the strategy for setting combination bits is as follows. For every pair of C_2-Common words appearing in a record, l_s combination bits will be set. (C_2 can be thought of as corresponding to C in the previous scheme.) In the new scheme, however, combination bits will also be set for pairs made from C_1-Common words and words with rank $C_2 + 1$ to C_3. A block descriptor will be of width $b_s + C_2$. For every C_2-Common word, an additional slice is allocated as in the previous scheme. Note that additional slices are not used for words with rank $C_2 + 1$ to C_3.

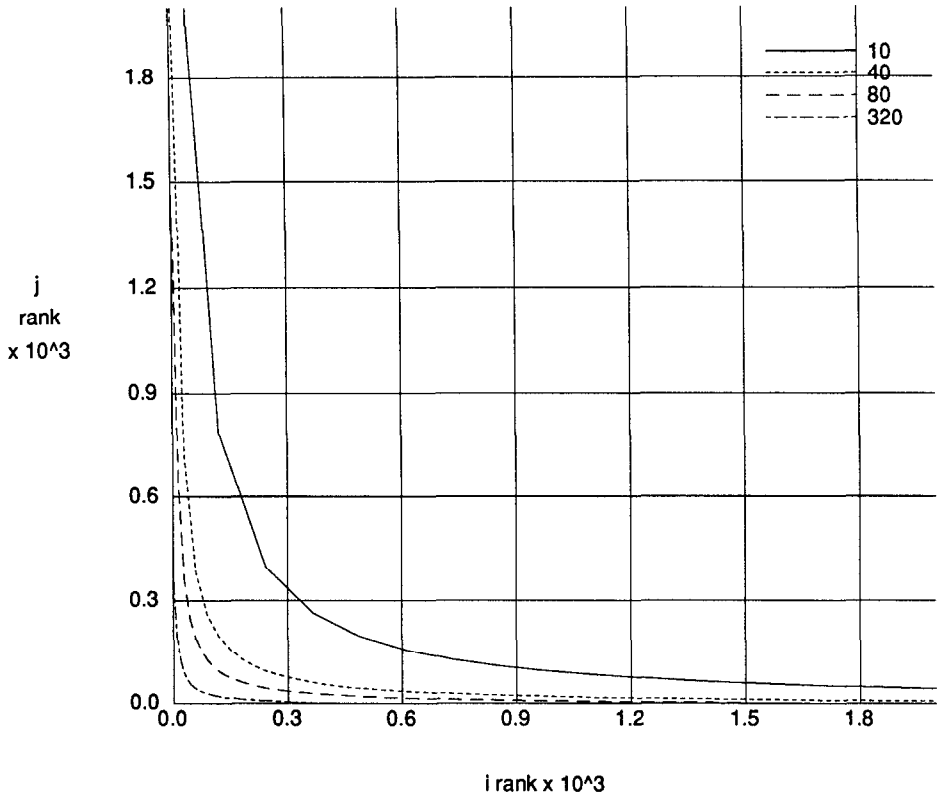

Fig. 9. Curves for which $\text{total}_{ij} = \Delta$, $\Delta = 10, 40, 80, 320$.

The new scheme can be analyzed in the same way as the previous scheme. With the new scheme, k_s bits are set in a block descriptor for each word with rank greater than C_2 that appears in that block. The probability that a bit is set by one of these words, μ_D, is then

$$\mu_D = 1 - \prod_{i=C_2+1}^{N_D} \left(1 - \frac{k_s}{b_s} \text{blk}_i\right).$$

The probability that a bit is set by a pair of words, μ_C, is now

$$\mu_C = 1 - \prod_{i=1}^{C_2-1} \prod_{j=i+1}^{C_2} \left(1 - \frac{l_s}{b_s} \text{blk}_{i,j}\right) \cdot \prod_{i=1}^{C_1} \prod_{j=C_2+1}^{C_3} \left(1 - \frac{l_s}{b_s} \text{blk}_{i,j}\right).$$

The number of unsuccessful block matches resulting from a query that specifies the values v_i and v_j, $i < j$, is now given by

$$\text{ubm}_{i,j} = \begin{cases} \mu^{l_s}(\text{total}_{i,j} - \text{true}_{i,j}), & \text{if } i, j \le C_2 \\ \mu^{l_s}(\text{total}_{i,j} - \text{true}_{i,j}), & \text{if } i \le C_1 \text{ and } C_2 + 1 \le j \le C_3 \\ \text{total}_{i,j} - \text{true}_{i,j}, & \text{otherwise.} \end{cases}$$

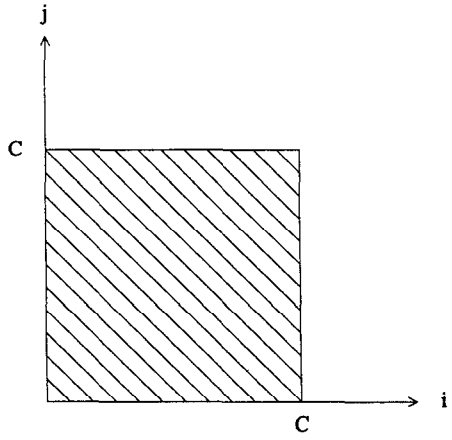

Fig. 10. Covered area when a single set of common words is used.

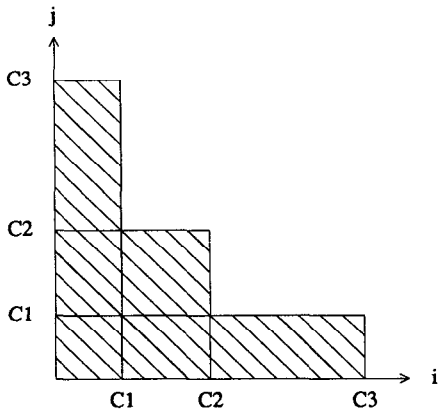

Fig. 11. Covered area when three sets of common words are used.

We now describe a method for choosing the parameters of the block descriptor file given data with a known distribution and a cutoff value, Δ. We will assume that the data can be parameterized by values α and β as in the previous section, and it is required to choose values for b_s, C_1, C_2, and C_3.

Step 1. Determine C_3. Compute the largest j such that $\text{total}_{1,j} \geq \Delta$.

Step 2. Determine C_2. There are two considerations we take into account when determining C_2. First, C_2 has to satisfy the requirement that $\text{total}_{C_2,C_2} \leq \Delta$, so we begin by computing the largest j such that $\text{total}_{j,j} \geq \Delta$. Next we consider whether it is possible to increase C_2 further, thereby covering a greater area, without incurring additional costs.

Consider the effect of the choice of C_2 on the density of the block descriptors. As we increase the number of C_2-Common words, fewer bits are set in the block descriptors by single values, since a C_2-Common word sets only a single bit in a block descriptor, whereas other words set k_s bits. On the other hand, as C_2 increases, more combination bits are set. We therefore attempt to estimate the largest value of C_2 such that the savings from single values are less than the costs due to combinational bits.

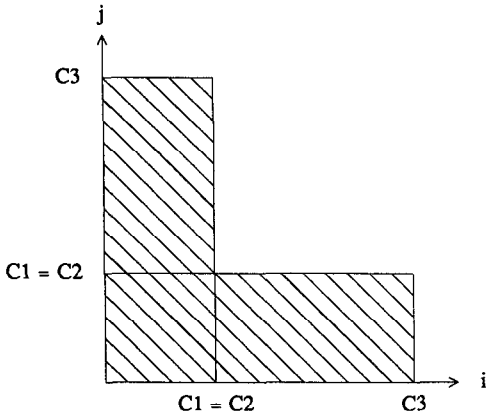

Fig. 12. Covered area when $C_1 = C_2$.

For a given C_2, the savings due to single values can be estimated as $k_s \cdot \sum_{i=1}^{C_2} \text{blk}_i - C_2$. The number of combination bits set per block descriptor by pairs of C_2-Common words is approximately $l_s \sum_{i=1}^{C_2-1} \sum_{j=i+1}^{C_2} \text{blk}_{i,j}$. We therefore compute the maximum C_2 such that

$$k_s \cdot \sum_{i=1}^{C_2} \text{blk}_i < l_s \sum_{i=1}^{C_2-1} \sum_{j=i+1}^{C_2} \text{blk}_{i,j} + C_2$$

and

$$\text{total}_{C_2,C_2} \leq \Delta.$$

Step 3. Determine C_1. In order to cover all pairs (i, j) for which the total block matches exceeds Δ, we require that

$$\text{total}_{C_1,C_2} \leq \Delta.$$

If $\text{total}_{C_2,C_2} = \Delta$, then $C_1 = C_2$ (Figure 12). Otherwise, $\text{total}_{C_2,C_2} < \Delta$, and we fine the greatest $j < C_2$ such that $\text{total}_{j,C_2} > \Delta$ (Figure 11).

Step 4. Determine b_s. With l_s combination bits set for pairs of common words, μ should be chosen according to the following formula:

$$\mu = \min\left[\left(\frac{1}{N_s}\right)^{1/k_s}, \left(\frac{1}{N_r - 1}\right)^{1/l_s}\right].$$

We propose a slight modification, however, since if $l_s = 1$ and N_r is large, this can impose an unnecessarily severe restriction on μ. The reason for the second constraint on μ is that, in the absence of combination bits, the number of unsuccessful block matches relative to the number of true matches approaches $N_r - 1$ for large i, j. We are really interested in only those values of $\text{ubm}_{i,j}/\text{true}_{i,j}$ for points (i, j) within the covered region. It is possible to determine the maximum value, f, of $\text{ubm}_{i,j}/\text{true}_{i,j}$ within this region by evaluating this function at the extreme points (C_1, C_3) and (C_2, C_2):

$$f = \max\left[\frac{\text{ubm}_{C_1,C_3}}{\text{true}_{C_1,C_3}}, \frac{\text{ubm}_{C_2,C_2}}{\text{true}_{C_2,C_2}}\right].$$

Then

$$\mu = \min\left[\left(\frac{1}{N_s}\right)^{1/k_s}, \left(\frac{1}{f}\right)^{1/l_s}\right].$$

The approach used to estimate what the value that b_s should be in order that the density of the block descriptors be μ is based on first estimating the average number of bits, #bits, that will be set in a block descriptor:

$$\#\text{bits} = \left(\sum_{i=C_2+1}^{N_D} \text{blk}_i\right) \cdot k_s + \left(\sum_{i=1}^{C_2-1} \sum_{j=i+1}^{C_2} \text{blk}_{i,j} + \sum_{i=1}^{C_1} \sum_{j=C_2+1}^{C_3} \text{blk}_{i,j}\right) \cdot l_s.$$

The first term represents the contribution of single words (words with rank greater than C_2) and the last two terms represent an estimate of the number of combinational bits. These sums can be computed using numerical quadrature techniques in order to reduce the computation time. In practice, the estimate of #bits would be obtained using sampling techniques, since most of the probabilities used in the preceding formula would not be known. Note that the preceding formula provides an overestimate of the number of bits set in a block descriptor. Based on these values of μ and #bits, we can then compute b_s from the formula

$$\mu \approx 1 - \left(1 - \frac{1}{b_s}\right)^{\#\text{bits}}.$$

6. COMPUTED RESULTS

In this section we present computed results using the model presented in the previous section. These computed results allow us to examine the effects of various parameters on system performance. Given a database of N records, each containing s terms to be indexed, we are interested in using the computed results to study the effects of varying N_s, the number of blocks, N_r, the number of records per block, the number of common words, the parameters of the block descriptors, b_s, k_s, l_s, and the parameters of the record descriptors, b_r and k_r. For given values of these parameters we want to know the expected query cost and the storage overhead generated.

We begin by considering various choices for the parameters previously given for a database that contains 1.5 million terms to be indexed, that is, $s \cdot N = s \cdot N_s \cdot N_r = 1.5$ million. We assume that the data are skew distributed as in Section 5, and that $p_i = \alpha/i^\beta$ with $\alpha = 0.08138$ and $\beta = 1$. With these choices of α and β, the number of distinct terms in the database is 122,066. In these tests, the parameters are chosen so that the expected number of false block matches per query is one, and the number of unsuccessful block matches per matching record, for queries which specify two or more common words, is limited to approximately one. In this case, μ is chosen according to the formula:

$$\mu = \min\left[\left(\frac{1}{N_s}\right)^{1/k_s}, \left(\frac{1}{f}\right)^{1/l_s}\right].$$

The numbers of C_i-Common words, $i = 1, 2, 3$ are calculated using the algorithm given in Section 5.

In presenting these results, we provide the value of $b_2 + C_2$, the width of a block descriptor, as well as a normalized measure of the overhead generated by the block descriptor file. The normalized measure given is the size of the block descriptor file divided by $s \cdot N$. This represents the overhead per indexed term.

The overhead generated by the block descriptor file represents typically 70–90 percent of the total overhead generated by the two-level scheme. If the record descriptors are designed so that they have approximately half their bits set (this is the most storage-efficient choice [29]), then b_r and k_r are related by the formula

$$b_r = \frac{k_r \cdot s}{\log_e 2}.$$

If these parameters are chosen so that the probability of a false record match is approximately $1/8N_r$, in order that there be only one false record match for every 8 block matches, then $k_r \approx 3 + \log_2 N_r$. A pointer is required for each record in the data file, so the total overhead generated by the record descriptor file per indexed term is $4.8 + 1.6 \log_2 N_r + \text{ptr}/s$, where ptr is the pointer size. Here we have used the fact that $1/\log_e 2 \approx 1.6$.

With these design decisions, the number of false record matches will be small and the performance of the two-level scheme will closely follow that predicted for Algorithm 2 of Section 4 of this paper. Assuming $N_s \leq P$, the cost of answering a query can be estimated as $k_s + n_s + n_r + 1$ disk accesses, where n_s is the total block matches (true block matches and unsuccessful block matches) and n_r is the total record matches (true record matches and false record matches). For single-term queries there will be no unsuccessful matches, while for multiterm queries, the number of unsuccessful block matches will be multiple, ϵ, of the true block matches, $0 \leq \epsilon \leq 1$. With the chosen parameters the number of false record matches will be approximately $n_s/8$, so $n_r \approx 9 \cdot n_s/8$. Since the performance of the method on query can be predicted, we will provide the storage overheads required in order to achieve these query times in the following tables.

In Figure 13, the effect that the parameters N_s, N_r, and l_s have on the storage overhead can be observed. When $l_s = 1$, and $N_r > 10$, the block descriptor density, μ, is severely constrained by the requirement that the number of unsuccessful block matches for queries that specify two or more common words be limited to one per matching record. The storage overhead is consequently very high. This restriction on μ virtually disappears when $l_2 = 2$. In this case,

$$\left(\frac{1}{f}\right)^{1/l_s} > \left(\frac{1}{N_s}\right)^{1/k_s}$$

and the limiting restriction on μ comes from the restriction on the number of false matches allowed. For each of the configurations, the number of C_2-Common words is small enough so that each C_2-Common word can be accommodated within main memory. The value for Δ in these tests was 50. In order to study the effect of the choice of Δ on system performance, a number of tests were run for which the only parameter to vary was Δ. The results of these tests appear in Figure 14. As Δ reduces in size, more queries are covered and the number of common words as well as the storage overhead rises. For the given data

	l_s	$\left(\frac{1}{N_s}\right)^{1/k_s}$	$\left(\frac{1}{f}\right)^{1/l_s}$	μ	C_1	C_2	C_3	$b_s + C_2$	Overhead
$N_s = 15{,}000$ $N_r = 10$	1	0.090	0.115	0.090	136	136	2436	4232	42.32
$N_s = 7500$ $N_r = 20$	1	0.107	0.055	0.055	191	191	2433	13,541	67.70
$N_s = 3000$ $N_r = 50$	1	0.135	0.022	0.022	294	294	2421	81,493	162.99
$N_s = 15{,}000$ $N_r = 10$	2	0.090	0.339	0.090	136	136	2436	6001	60.01
$N_s = 7500$ $N_r = 20$	2	0.107	0.235	0.107	191	191	2433	9825	49.13
$N_s = 3000$ $N_r = 50$	2	0.135	0.148	0.135	294	294	2421	18,538	37.08

Fig. 13. Results for $s = 10$, $k_s = 4$, $\Delta = 50$.

	l_s	Δ	$\left(\frac{1}{N_s}\right)^{1/k_s}$	$\left(\frac{1}{f}\right)^{1/l_s}$	μ	C_1	C_2	C_3	$b_s + C_2$	Overhead
$N_s = 15{,}000$	1	∞	0.090	—	0.090	1	1	1	3432	34.32
$N_r = 10$	1	100	0.090	0.115	0.090	95	95	1216	4010	40.10
$s = 10$	1	50	0.090	0.115	0.090	136	136	2436	4232	42.32
$k_s = 4$	1	10	0.090	0.113	0.090	311	311	12,200	4925	49.25
$N_s = 7500$	2	∞	0.107	—	0.107	1	1	1	5337	26.68
$N_r = 20$	2	100	0.107	0.237	0.107	132	132	1212	8930	44.65
$s = 10$	2	50	0.107	0.235	0.107	191	191	2433	9825	49.13
$k_s = 4$	2	10	0.107	0.232	0.107	437	437	12,198	12,327	61.63
$N_s = 3000$	2	∞	0.135	—	0.135	1	1	1	9441	18.88
$N_r = 50$	2	100	0.135	0.151	0.135	201	201	1200	16,725	33.45
$s = 10$	2	50	0.135	0.148	0.135	294	294	2421	18,538	37.08
$k_s = 4$	2	10	0.135	0.145	0.135	684	684	12,187	23,528	47.06

Fig. 14. Effect of Δ on various databases.

distribution, the number of C_3-Common words was less than 10 percent of the number of unique terms even for $\Delta = 10$.

In order to reduce the storage overhead generated by the block descriptor file, various alternatives exist. The restrictions on μ can be relaxed by allowing more false matches or more unsuccessful block matches. Another possibility is to increase k_s and l_s, and thereby increase the query cost by only a constant number of disk accesses. For the various combinations of N_s and N_r given in Figure 14, the storage overheads generated using values of k_s ranging from 4 to 12 are presented in Figure 15. For each value of k_s the value of l_s within the range of 1 to 4 that results in the smallest overhead was computed. In these tests Δ was fixed at 50. It can be observed that for these databases of 1.5 million tokens, storage overheads of between 20 and 25 bits per indexed term are feasible.

$N_s = 15{,}000$ $N_r = 10$				
k_s	l_s	μ	$b_s + C_2$	Overhead
4	1	0.090	4232	42.32
6	2	0.201	3096	30.96
8	3	0.301	2774	27.74
10	3	0.383	2323	23.23
12	3	0.449	2090	20.90
$N_s = 7500$ $N_r = 20$				
k_s	l_s	μ	$b_s + C_2$	Overhead
4	2	0.107	9825	49.13
6	3	0.226	6603	33.02
8	3	0.328	4853	24.27
10	3	0.381	4494	22.47
12	4	0.476	4234	21.17
$N_s = 3000$ $N_r = 50$				
k_s	l_s	μ	$b_s + C_2$	Overhead
4	2	0.135	18,538	37.08
6	3	0.263	13,293	26.59
8	4	0.368	11,853	23.71
10	4	0.385	12,176	24.35
12	4	0.385	13,173	26.35

Fig. 15. Effect of k_s and l_s on storage overhead.

$N_s = 15{,}000$ $N_r = 10$								
s	$tokens(s \cdot N)$	α	l_s	C_1	C_2	C_3	μ	$b_s + C_2$	Overhead
2	300,000	0.092	3	23	42	474	0.303	359	17.95
5	750,000	0.085	2	68	76	1271	0.301	953	19.06
10	1,500,000	0.081	2	136	136	2436	0.301	2306	23.06
20	3,000,000	0.077	2	260	260	4632	0.301	6224	31.12
50	7,500,000	0.072	2	610	610	10,864	0.301	26,895	53.79

Fig. 16. Effect of increasing s while N_s and N_r remain fixed.

In order to study the effect of s, Figure 16 presents results for which the number of records, N, is fixed at 150,000, with $N_s = 15{,}000$ and $N_r = 10$, while s is increased from 2 to 50. As s increases, so does the number of indexed terms, $s \cdot N$. For each database, the value of β is fixed at 1, while α is chosen so that the token of lowest rank appears once in the database ($s \cdot N \cdot p_{N_D} = 1$). The value of k_s is fixed at 8 and $\Delta = 50$. For small values of s the overhead generated is very small. For $s > 5$, μ is fixed at 0.301 and is constrained by the requirement that the number of false matches be limited to one. The storage overhead increases with s and the only way to reduce storage costs substantially is to relax this restriction. Note that although the number of indexed terms increases only linearly with s, the number of combinational bits will grow faster than linearly.

$N_s = 40{,}320$ $N_r = 26$ $s = 20$				
k_s	l_s	μ	$b_s + C_2$	Overhead
4	2	0.071	69,744	134.12
6	2	0.171	30,583	58.81
8	3	0.266	27,024	52.08
10	3	0.344	21,302	40.97
12	4	0.413	21,777	41.88
14	4	0.449	20,423	39.27

Fig. 17. Database of 1,048,576 records, each containing 20 terms.

$N_s = 15{,}000$ $N_r = 10$ $s = 10$									
α	β	N_D	C_1	C_2	C_3	l_s	μ	$b_s + C_2$	Overhead
0.017	0.8	322,676	66	66	1889	2	0.301	2190	21.90
0.081	1.0	122,066	136	36	2436	2	0.301	2306	23.06
0.201	1.2	36,786	127	127	1410	2	0.301	1866	18.66
$N_s = 40{,}000$ $N_r = 25$ $s = 20$									
α	β	N_D	C_1	C_2	C_3	l_s	μ	$b_s + C_2$	Overhead
0.010	0.8	4,175,795	302	302	13,070	3	0.266	14,195	28.39
0.068	1.0	1,360,548	663	663	13,588	3	0.266	23,378	46.76
0.193	1.2	307,664	534	534	6621	2	0.207	18,698	37.40

Fig. 18. Effect of vary the skew of the data, β.

Results for a large database are presented in Figure 17. This database contains over one million records, each containing 20 attribute values. The parameters N_s and N_r have the same values as were used in the database described in Section 4. For each value of k_s, l_s is chosen from the range 1 to 4 such that storage costs are minimized, and Δ is fixed at 50. For this database of approximately 20 million tokens, storage overheads of around 40 bits per indexed term are feasible.

In the results presented so far, it has been assumed that the data are Zipf distributed with a skew of 1.0. The effect of varying the skew parameter, β, is presented in Figure 18. In these tables, results are presented for a database of 150,000 records as well as for a database of 1,000,000 records. The parameter β varies from 0.8 to 1.2. As in the previous table, Δ is fixed at 50 and k_s is fixed at 8. As β decreases, the data become more uniformly distributed and for realistic values of β, good results are possible. As β increases, the number of common words required decreases, and for the databases in Figure 18, the number of combination bits required decreases.

7. EXPERIMENTAL RESULTS

An experiment was conducted using a library database of 150,000 records, each containing approximately 20 terms to be indexed ($s \cdot N \approx 3$ million). Each record contains information about books in an institute library. The terms include author names, subject and title keywords and phrases, publication dates, and

accession numbers (unique identifiers). These records were stored in a database using the two-level scheme with $N_s = 16{,}384$ and $N_r = 11$.

In designing this database, we chose $C_1 = C_2 = 300$ and $C_3 = 2000$. The most common term v_1 (title keyword = editor) appeared in approximately 7000 blocks, v_{10} (publication date = 1970) appeared in 4220 blocks, v_{300} (title keyword = model) appeared in 903 blocks, and v_{2000} appeared in 58 blocks. With the above choices for C_1 to C_3, combination bits would be set for multiterm queries involving more than 50 block matches (i.e., $\Delta \approx 50$), since

$$\begin{aligned}\Delta &\geq \max(N_s \text{blk}_{C_1,C_3},\ N_s \text{blk}_{C_2,C_2}) \\ &= \max(7000 \cdot 58/16{,}384,\ 903 \cdot 903/16{,}384) \\ &= \max(24.8,\ 49.8) \approx 50.\end{aligned}$$

The characteristics of the records were as follows. The average number of terms per record was 19.97, while the average number of distinct terms per record was 13.29. This included 3.94 C_1-Common words, 6.67 C_3-Common words, and 6.62 regular words.

At the block level, the average number of records per block was 9.04. The average number of distinct terms per block was 112.59. This included 33.39 C_1-Common words, 56.48 C_3-Common words, and 56.10 regular words.

The database parameters were as follows:

$$b_s = 10{,}700, \quad k_s = 4, \quad \text{and} \quad l_s = 1.$$
$$b_r = 320 \quad \text{and} \quad k_r = 8.$$

This represents a storage overhead of 7.4 bytes per indexed term due to the block descriptor file and 2.2 bytes per indexed term due to the record descriptor file. With these parameters the average density of the block descriptor file μ was 0.06. The average number of bits set in positions 1 through b_s of a block descriptor by individual terms was 317. The average number of combination bits per block descriptor was 267. In addition, an adjacency bit was set for each pair of adjacent terms in a subject or title phrase. The average number of adjacency bits per block descriptor was 73. Note that b_s was chosen to be large (the expected number of false block matches for single term queries was 0.22) in order to study the effect of unsuccessful block matches. As a consequence, we have assumed that any failed block matches are due to unsuccessful block matches.

For comparison, a second database was created with the same descriptor sizes (storage overheads) but with no common words ($C_1 = C_2 = C_3 = 0$).

In order to conduct the experiment 1000 "random" queries using terms contained in the database were generated. Each query contained two terms. The queries were executed and various parameters were recorded. For 375 of these queries, combination bits were generated using the database with 2000 common words. For each of these queries the number of unsuccessful block matches was recorded and compared to the number of unsuccessful block matches that occurred using the database with no common words. The results appear in Figures 19 and 20. In Figure 19 the average number of unsuccessful block matches per record match is plotted against the number of matching records for each of the 375 queries. In Figure 20 the absolute numbers of unsuccessful matches is plotted against the number of matching records for each of the two methods. It

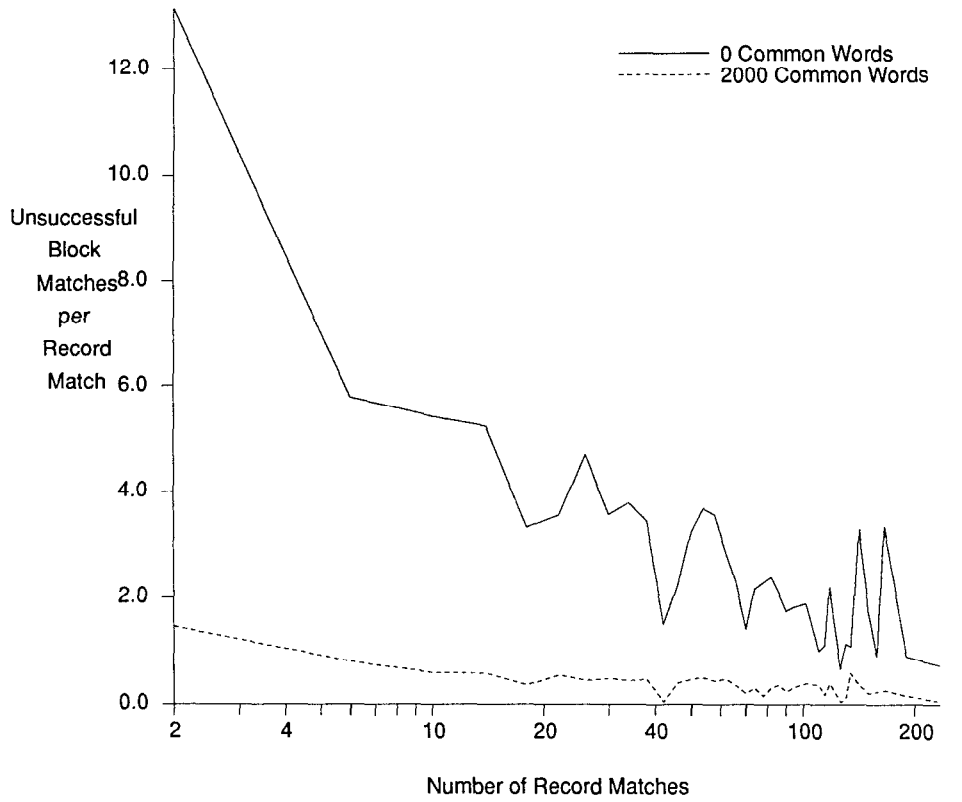

Fig. 19. Average unsuccessful block matches per record match, $b_s = 10{,}700$, $k_s = 4$, $l_s = 1$.

can be seen that the use of a single combination bit has the desired effect of eliminating most of the unsuccessful block matches. For the database without common words we would expect that the number of unsuccessful block matches per true record match would be approximately 10 (i.e., $N_r - 1$) when the number of record matches was small and we would expect this to reduce to less than one for the database with 2000 common terms. These trends were reflected in the experimental results.

A second experiment was conducted in order to test the effect of setting adjacency bits when forming the block descriptors. The subject and title descriptors contained in the records consisted of word phrases as well as single words. For each pair of adjacent words contained in a phrase, a single adjacency bit was set in the appropriate block descriptor. The cost of setting these bits is small. These bits contributed less than 12 percent to the density of the block descriptor file. Again, 1000 queries were constructed. Each query contained a phrase consisting of two words. These queries were executed using two methods. In the first method, the adjacency bit was used to reduce the number of failed block matches. In this case, a failed match can occur even when a record contains the two terms, if they are not in adjacent locations. In the second method, the adjacency bit was not set in the query descriptor. The results are presented in

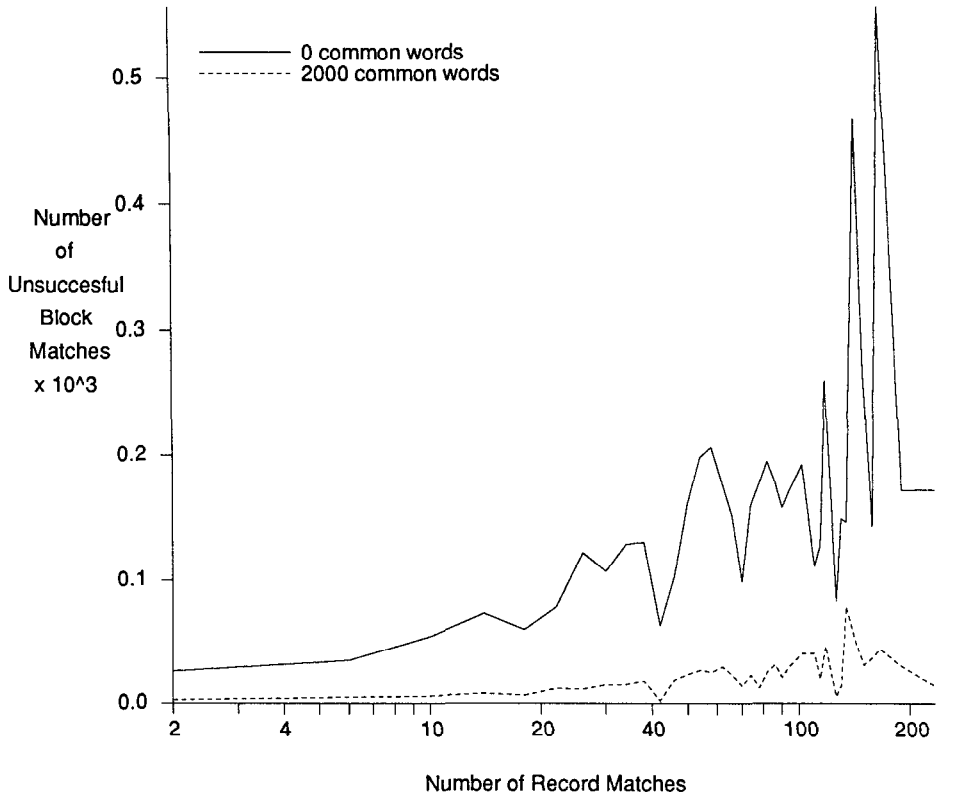

Fig. 20. Unsuccessful block matches vs. record matches, $b_s = 10{,}700$, $k_s = 4$, $l_s = 1$.

Figure 21. The results are impressive and indicate that direct indexing on word phrases at low cost is possible using descriptor-based methods.

In order to study the effect of the algorithm using parameters for which the storage overheads are low, the experiments were repeated using the parameters $b_s = 6000$, $k_s = 6$, and $l_s = 2$. With these parameters, the storage overhead due to the block descriptor file is reduced from 7.4 to 4.1 bytes per term. The configuration at the record descriptor level remains unchanged. In Figure 22 the results for the two sets of block descriptor parameters are presented. The results obtained using the smaller block descriptor file are very similar to those obtained previously and show that the effect of reducing b_s can be compensated by increasing k_s and l_s appropriately.

8. IMPLEMENTATION ISSUES

In order to implement the scheme proposed in Section 5, it is necessary to identify the common terms appearing in the data file. Sampling techniques must be used to determine these common words together with their frequencies. The results computed from the theoretical model show that although the number of C_2-Common words will typically be of the order of several hundred, the number of

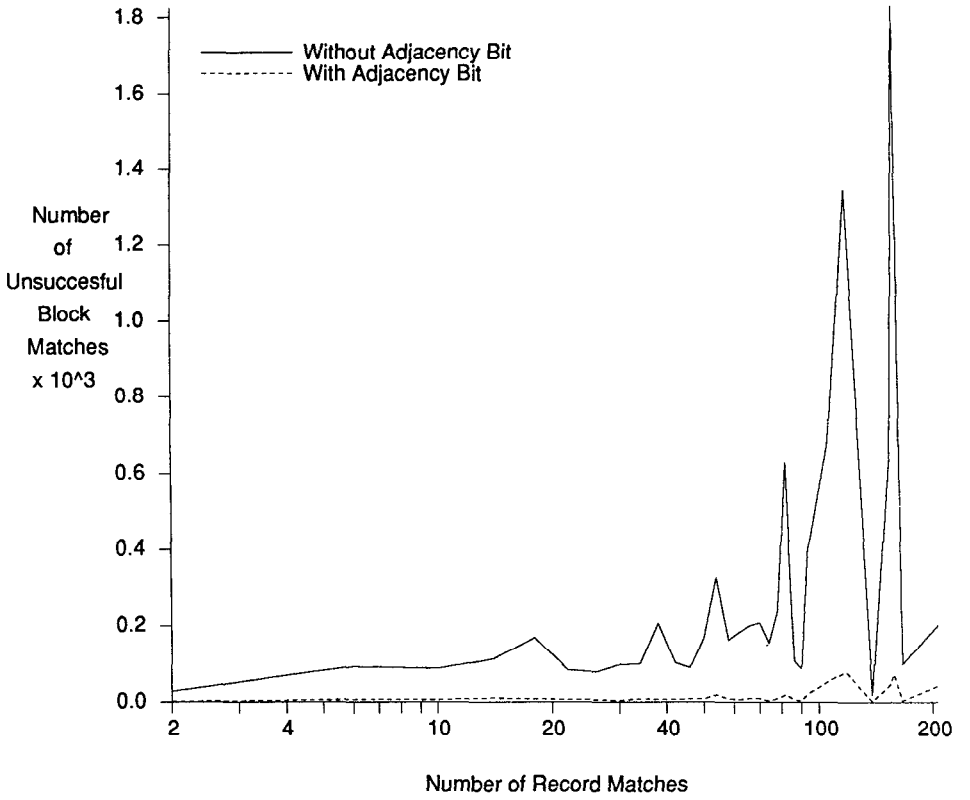

Fig. 21. Effect of adjacency bits on unsuccessful block matches.

C_3-Common words may be several thousand. During execution of the two-level method, it may not be feasible to store each C_3-Common word within main memory. The approach we have taken is to store each C_2-Common word explicitly and to use a filter to identify the C_3-Common words. We briefly outline a method that we have found useful for representing the C_3-Common words with the use of a filter. The filter is just a bit vector that is formed using hash-coding techniques similar to those used to form record descriptors.

Before the filter can be formed it is necessary to determine the common terms and their frequencies. To reduce the number of terms that have to be stored in memory during the sampling process, the following technique can be used. A descriptor of length b with k bits set is formed for each term in the database. An array of b integers is used to keep a cumulative count of the number of times that each bit position is set by one of these terms.

During a second pass of the data file, a table of terms and their frequencies is formed. For each term the minimum of the counts for each of the k bits set by the term is also stored. The minimum count provides a rough indication of the rank of the term. The way terms are added to the table is as follows. For each term examined, the table is searched and if the term already exists in the table, its frequency is incremented. Otherwise, we attempt to add the term to the table.

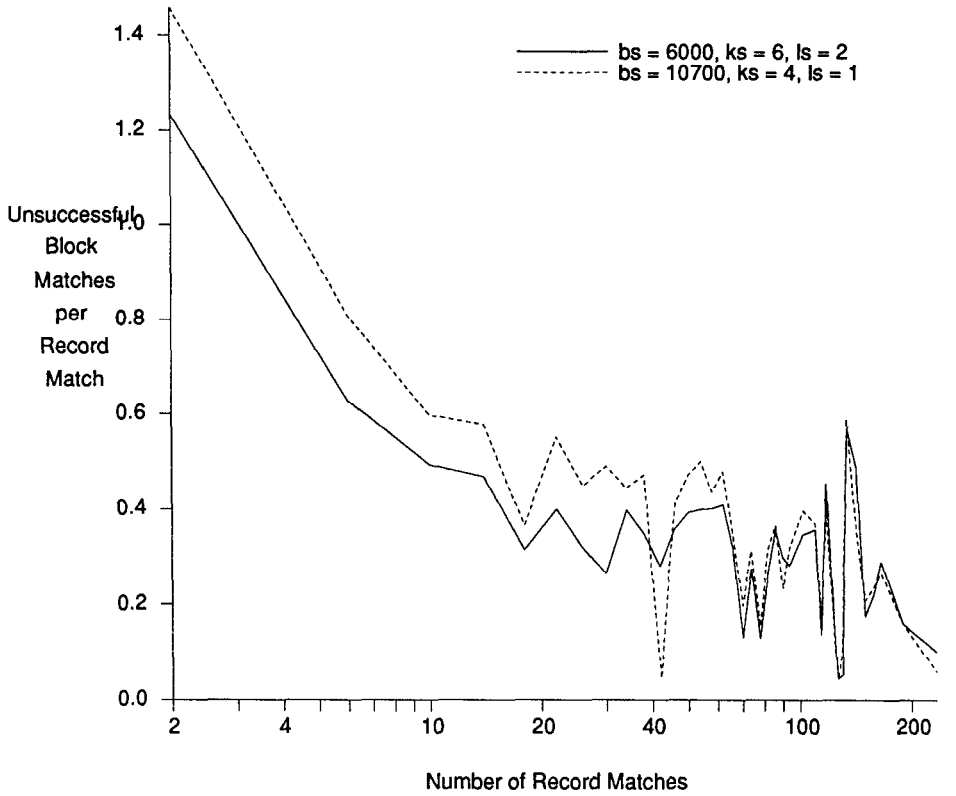

Fig. 22. Comparing databases with 2000 common words and different block descriptor parameters.

If the table is full, the count for the new term is compared with the lowest count in the table. If the new term count is greater, then the new term replaces the term associated with the lowest count. The table size should be chosen sufficiently large to include all C_3 terms, the number of which is not generally known at the beginning of the pass. Even when C_3 is known, the table size should be larger than C_3, as the sampling method may cause some less frequent terms to be accepted as more frequent terms. A table size of 5000 should be adequate for most applications.

From the frequencies of the terms in the table, we can determine C_1, C_2, and C_3 using the steps described in Section 5.

Once the C_3-Common terms are known, a filter can be constructed by simply superimposing all of the term descriptors for the common terms. This type of filter is known as a Bloom filter [3]. In order to determine whether an arbitrary term appearing in a record is a common term, its descriptor is formed. If every bit set in the descriptor is also set in the filter, the term is deemed to have passed through the filter and is designated a common term.

This filtering technique has been used on the library database described in the previous section. Filters were formed using parameters $b = 10{,}700$ and $k = 4$. In order to test the effectiveness of the filter, a large number of terms were randomly

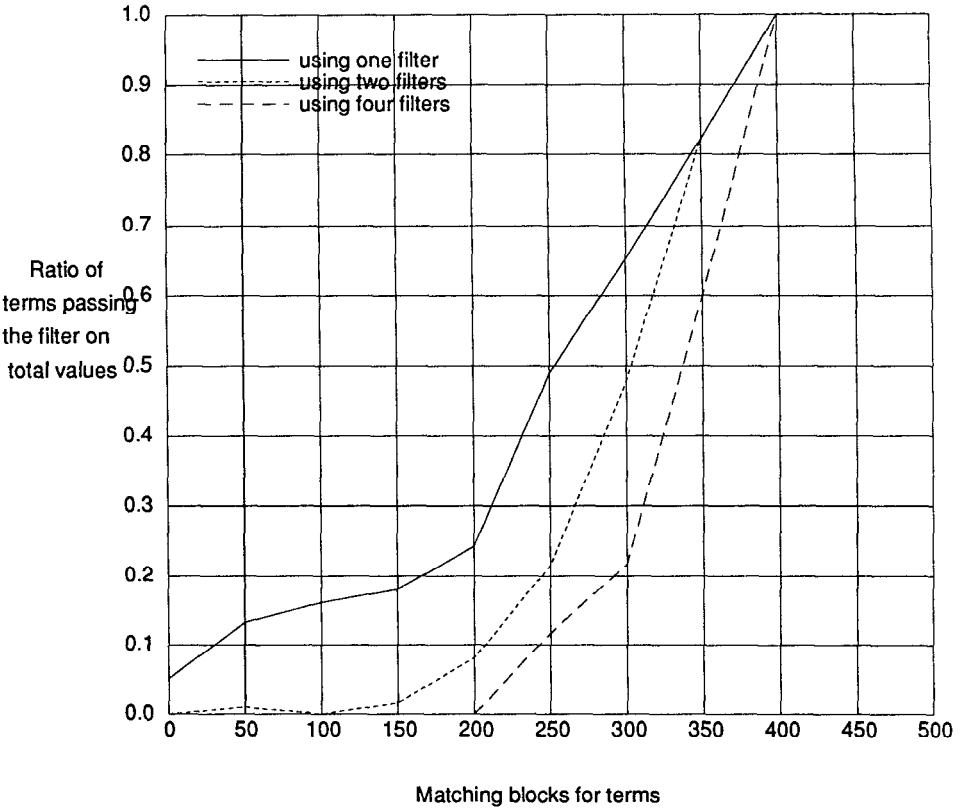

Fig. 23. Ratio of terms passing through the filter, $b = 10{,}700$, $k = 4$.

selected, and for each term, the number of matching blocks was computed. It was then determined whether the term had passed through the filter. Results obtained using a filter formed from terms that appeared in 400 or more blocks appear in Figure 23 (refer to the solid line). In addition to the terms that appeared in more than 400 blocks, a large number of less frequently occurring terms passed through the filter. For example, approximately 20 percent of the terms that occurred in 160 blocks passed through the filter. If a single such filter was used to identify C_3-Common words, then the C_3-Common words would include some infrequently occurring terms. This would result in some unnecessary combination bits being set, but would not detract very much from the efficiency of the scheme.

In order to reduce the number of infrequently occurring terms that are designated as C_3-Common words, several strategies are possible. One approach is to use more than one filter. A word is then designated a C_3-Common word only if it passes through all the filters. Figure 23 presents the results when two filters are used and when four filters are used. The extra filters were constructed using the same values for b and k. When four filters were used, virtually no word that appeared in less than 200 blocks passed through all of the filters.

An alternative to this technique for building a filter was used by McIlroy [18]. After the C_3-Common words have been determined a filter constructed with very

large b is constructed and the resulting bit string is compressed. This method saves space [11] but is slower on query.

The following method can be used to determine the database parameters when setting up a database.

(1) Sample the data to form the array of b integers that will be used for identifying the C_3-Common words. At the same time compute the number of unique terms per record, s.

(2) Determine N_s and N_r. The parameters at the record descriptor level are determined by the following three equations.

$$b_r = k_r \cdot \frac{s}{\log_e 2},$$

$$F_r = \left[\frac{1}{2}\right]^{k_r},$$

$$P = N_r \cdot (b_r + \text{ptr}).$$

The first equation results from the requirement that the density of the record descriptor be 1/2 The second equation gives the probability of a false record match in terms of k_r; a typical design would set the value of F_r to something like $1/(4N_r)$. The third equation results from the requirement that all of the record descriptors fit on a single page of size P. The user supplies the values of F_r; s and P are known and the above system of three implicit equations is solved for k_r, b_r, and N_r. The value of N_s can then be determined using $N = N_r \cdot N_s$.

(3) The data are resampled and the array of b integers from step 1 is used to determine the common words as previously described. At the same time the frequencies of these common words at both the record and block level are computed. In addition, the number of unique terms per block, s_B can be computed.

(4) C_1, C_2, and C_3 are computed using the methods outlined in steps 1 to 3 of the algorithm given in Section 5.

(5) This is a modified version of step 4 of the algorithm used in Section 5. The required bit density, μ, of the block descriptor file can be computed using the formula

$$\mu = \min\left[\left(\frac{1}{N_s}\right)^{1/k_s}, \left(\frac{1}{N_r - 1}\right)^{1/l_s}\right].$$

The choices of k_s and l_s will affect the query times, storage overheads as well as the interactive insertion costs so they are application dependent. A number of different values for these parameters should be considered and the various tradeoffs analyzed. The number of (nonunique) bits, #bits, that will be set in a block descriptor formula can be estimated using the formula

$$\#\text{bits} = \left(s_B - \sum_{i=1}^{C_2-1} \text{blk}_i\right) \cdot k_s + \left(\sum_{i=1}^{C_2-1} \sum_{j=i+1}^{C_2} \text{blk}_{i,j} + \sum_{i=1}^{C_1} \sum_{j=C_2+1}^{C_3} \text{blk}_{i,j}\right) \cdot l_s.$$

Here, the probabilities, blk_i, $i = 1, \ldots, C_3$ are those computed from step 3. Alternatively, the data can be sampled a third time to obtain an estimate of #bits. The block descriptor parameter, b_s, can then be computed using

$$\mu \approx 1 - \left(1 - \frac{1}{b_s}\right)^{\#\text{bits}}.$$

9. A FAST INSERTION ALGORITHM

By far the major cost when inserting a record comes from updating the corresponding block descriptor, since the block descriptor file is stored in bit slice form. For each of the terms specified in a record, k_s bits of the block descriptor will need to be set and these bits will typically reside on separate pages of physical store. Thus for each term, k_s reads and possibly k_s writes must be performed. If there are s terms in the record, then up to $2 \cdot s \cdot k_s$ disk accesses will need to be performed. Actually a slightly smaller number than this is generally required. If a particular bit position has already been set by another term, then the write can be avoided and one disk access saved. Ignoring this factor, the number of disk accesses required to form the block descriptor file when N records are inserted into the database is $2 \cdot N \cdot s \cdot k_s$.

The other insertion costs, namely writing the record descriptor and the data to disk, involve only two read/write pairs per record. In the following, when we refer to the insertion costs, we are referring to the costs involved in forming the block descriptor file.

For interactive insertions, the costs cannot be substantially reduced. If, however, the database is initially loaded using a batch insertion facility, considerable savings can be effected. Because the block descriptor file is a two-dimensional bit matrix, one strategy is to form parts of this matrix in a buffer in memory and then write these formed submatrices on to disk in such a way that no parts of these submatrices need be overwritten at a later time.

It is assumed that a buffer is available in memory; the size of the buffer required can be as small as 100K bytes (depending on the database parameters), but larger buffer sizes will result in more efficient insertion. Let us assume that the buffer size is B (bits) and let $M = \lfloor B/b_s \rfloor$. In this case, the buffer can hold M block descriptors. Records are then processed in groups of $M \cdot N_r$, and these records are allocated to the next available M blocks. The block descriptors for these records are formed in memory and after all the records from this group have been processed, the b_s partial bit slices of size M are written to disk (see Figure 24). This requires b_s read/write pairs each time the buffer is written to disk (although only writes are required on the first pass). The total number of disk accesses to load N records using this method is therefore $(1 + 2 \cdot (\lceil N_s/M \rceil - 1)) \cdot b_s$.

Consider the following database of approximately one million records, each containing 20 values:

$N = 1{,}048{,}576$	Number of records,
$s = 20$	Values per record,
$P = 8192$	Page size, 1K bytes,
$N_s = 40{,}960$	Number of blocks

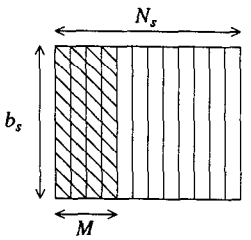

Fig. 24. In the one-pass algorithm, b_s partial slices of length M are written to disk.

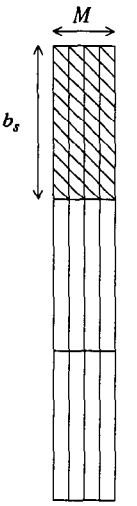

Fig. 25. Temporary file created during two-pass algorithm.

$N_r = 26$	Records per block
$b_s = 20{,}185$	Block descriptor width
$k_s = 4$	Bits set per indexed value
$B = 2{,}097{,}152$	Buffer size, 256K bytes

The cost of inserting a record using the interactive method is approximately $2 \cdot s \cdot k_s = 160$ disk accesses, while the cost per record using the batch insertion scheme is only 15.30 disk accesses, a saving of over tenfold.

Further improvements to the batch insertion scheme can be made if two block descriptor files can be temporarily accommodated on disk. With this method, the descriptors formed in memory are written contiguously onto the first file in bit slices of length M (see Figure 25). After all the block descriptors have been formed in the first file, they are rearranged onto the second file. Suppose that m bit slices of length N_s can be stored in the buffer, that is, $m = \lfloor B/N_s \rfloor$, then the rearrangement proceeds as follows. The appropriate bit slices from the first file are collected and reorganized to form m complete slices (see Figure 26). The m complete slices are then written to disk and the process is repeated until the reorganization is complete. Each step requires that $\lceil N_s/M \rceil$ sets of $m \cdot M$ consecutive bits be read from the first file in order to form m complete slices in memory. These slices are written to the second file, requiring $\lceil (m \cdot N_s)/P \rceil$ writes (and perhaps a small number of reads). The number of disk accesses required to

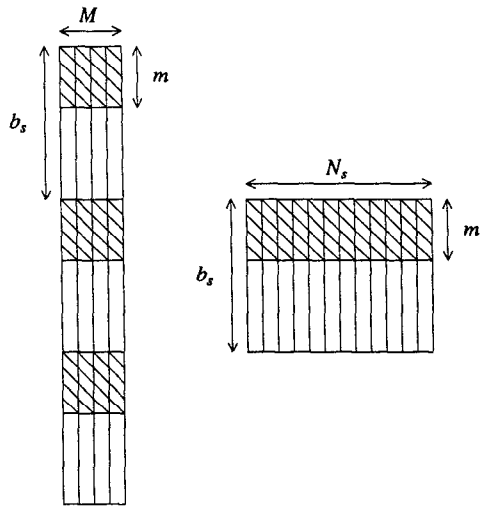

Fig. 26. Construction of the block descriptor file from the temporary file.

read $m \cdot M$ consecutive bits is $f(m, M, P)$, where $f(m, M, P) = k + 1$ if $m \cdot M = (k + 1/2^j) \cdot P$ for some integers j, $k \geq 0$ and $f(m, M, P) \approx 1 + m \cdot M/P$ otherwise. The formula for $f(m, M, P)$ recognizes that the $m \cdot M$ consecutive bits to be read may cross a page boundary. The number of steps required in the reorganization process is $\lceil b_s/m \rceil$. The cost to initially form the first file is $\lceil (b_s \cdot M)/P \rceil \cdot \lceil N_s/M \rceil$. The total cost is therefore

$$\left\lceil \frac{b_s \cdot M}{P} \right\rceil \cdot \left\lceil \frac{N_s}{M} \right\rceil + \left\lceil \frac{b_s}{m} \right\rceil \cdot \left\{ f(m, M, P) \cdot \left\lceil \frac{N_s}{M} \right\rceil + \left\lceil \frac{m \cdot N_s}{P} \right\rceil \right\}.$$

Using the facts that $m \cdot N_s \approx M \cdot b_s \approx B$ and $N_s/M \approx b_s/m$, we can express the total cost as

$$\left\lceil \frac{N_s}{M} \right\rceil \cdot \left\{ 2 \left\lceil \frac{B}{P} \right\rceil + f(m, M, P) \cdot \left\lceil \frac{N_s}{M} \right\rceil \right\}.$$

For the example database described earlier in this section, we have $M = 103$, $m = 51$, $\lceil N_s/M \rceil = 398$, $f(m, M, P) = 1.64$, and the insertion cost per record is only 0.44 disk accesses. This is approximately 400 times cheaper than the interactive method and 40 times cheaper than the previous batch insertion method.

10. CONCLUSIONS

For large data files, the one-level implementations of descriptor file methods become relatively inefficient because of the large amount of descriptor file that has to be examined at query time. A two-level scheme overcomes this problem, but introduces a new cost due to unsuccessful block matches. One way to reduce the cost of these unsuccessful block matches is to set combinational bits in the block descriptors. This approach can be extended so that direct indexing of word phrases is supported. Both theoretical and experimental results indicate that the use of combinational bits significantly reduces query costs, and the storage

overheads incurred are relatively small. It is also possible to achieve very low insertion costs with the two-level method if insertions are batched.

ACKNOWLEDGMENTS

We wish to thank the referees for their valuable suggestions and Professor G. Salton for his helpful comments regarding this paper.

REFERENCES

1. AHO, A. V., AND CORASICK, M. J. Fast pattern matching: An aid to bibliographic search. *Commun. ACM 18*, 6 (1975), 333–340.
2. BESAI, B. C., GOYAL, P., AND SADRI, F. A data model for use with formatted and textual data. *J. Am. Soc. Inf. Sci. 37*, 3 (May 1986), 158–165.
3. BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*, 7 (July 1970), 422–426.
4. BOYER, R. S., AND MOORE, J. S. A fast string searching algorithm. *Commun. ACM 20*, 10 (1977), 762–772.
5. CHRISTODOULAKIS, S. Implications of certain assumptions in database performance evaluation. *ACM Trans. Database Syst. 9*, 2 (June 1984), 163–186.
6. CHRISTODOULAKIS, S., AND FALOUTSOS, C. Design considerations for a message file server. *IEEE Trans. Softw. Eng. SE-10*, 2 (1984), 201–210.
7. CHRISTODOULAKIS, S. Issues in the architecture of a document archiver using optical disk technology. In *Proceedings of the International SIGMOD Conference on Management of Data* (Austin, Tex., May 28–31, 1985). ACM, New York, 1985, pp. 34–50.
8. COLOMB, R. M., AND JAYASOORIAH, J. A clause indexing system for PROLOG based on superimposed coding. *Aust. Comput. J. 18*, 1 (Feb. 1986), 18–25.
9. DATTOLA, R. FIRST: Flexible information retrieval system for text. *J. Am. Soc. Inf. Sci. 30*, (1979), 9–14.
10. FALOUTSOS, C. Access methods for text. *Comput. Surv. 17*, 1 (Mar. 1985), 49–74.
11. FALOUTSOS, C. Signature files: Design and performance comparison of some signature extraction methods. In *Proceedings of the International SIGMOD Conference on the Management of Data* (Austin, Tex., May 28–30, 1985). ACM, New York, 1985, pp. 63–82.
12. FALOUTSOS, C., AND CHRISTODOULAKIS, S. Design of a signature file method that accounts for nonuniform occurrence and query frequencies. In *Proceedings of the 11th Conference on Very Large Data Bases* (Stockholm, Aug. 21–23, 1985). pp. 165–170.
13. HARRISON, M. C. Implementation of the substring test by hashing. *Commun. ACM 14*, 12 (Dec. 1971), 777–779.
14. HASKIN, R. L. Special purpose processors for text retrieval. *Database Eng. 4*, 1 (1981), 16–29.
15. HASKIN, R. L., AND LORIE, R. A. On extending the functions of a relational database system. In *Proceedings of the International SIGMOD Conference* (Orlando, Fla., June 2–4, 1982). ACM, New York, 1982, pp. 207–212.
16. HILLIS, W. D. *The Connection Machine*. M.I.T. Press, Cambridge, Mass., 1985.
17. KNUTH, D. E., MORRIS, J. H., AND PRATT, V. R. Fast pattern matching in strings. *SIAM J. Comput. 6*, 2 (1977), 323–350.
18. MCILROY, M. D. Development of a spelling list. *IEEE Trans. Commun. COM-30*, 1 (Jan. 1982), 91–99.
19. MCLEOD, I. A. A database management system for document retrieval applications. *Inf. Syst. 6*, 2 (1981), 131–137.
20. PFALTZ, J. L., BERMAN, W. J., AND CAGLEY, E. M. Partial-match retrieval using indexed descriptor files. *Commun. ACM 23*, 9 (Sept. 1980), 522–528.
21. RABITTI, F., AND ZIZKA, J. Evaluation of access methods to text documents in office systems. In *Proceedings of the 3rd Joint ACM-BCS Symposium on Research and Development in Information Retrieval* (Cambridge, Mass., July 2–6, 1984). pp. 21–40.

22. RAMAMOHANARAO, K., AND SHEPHERD, J. A superimposed codeword indexing scheme for very large Prolog databases. In *Proceedings of the Third International Conference on Logic Programming* (London, 1986). pp. 569–576.
23. ROBERTS, C. S. Partial-match retrieval via the method of superimposed codes. *Proc. IEEE 67*, 12 (Dec. 1979), 1624–1642.
24. SACKS-DAVIS, R., AND RAMAMOHANARAO, K. A two level superimposed coding scheme for partial match retrieval. *Inf. Syst. 8*, 4 (1983), 273–280.
25. SACKS-DAVIS, R. Performance of a multi-key access method based on descriptors and superimposed coding techniques. *Inf. Syst. 10*, 4 (1985), 391–403.
26. SALTON, G. *The SMART Retrieval System—Experiments in Automatic Document Processing.* Prentice-Hall, Englewood Cliffs, N.J., 1971.
27. SALTON, G., AND MCGILL, M. J. *Introduction to Modern Information Retrieval.* New York, McGraw-Hill, 1983.
28. STAIRS/VS REFERENCE MANUAL. *IBM System Manual*, 1979.
29. STIASSNY, S. Mathematical analysis of various superimposed coding schemes. *Am. Documen. 11*, 2 (Feb. 1960), 155–169.
30. VAN-RIJSBERGEN, C. J. *Information Retrieval*, 2nd. ed. Butterworths, London, 1979.
31. WHANG, K.-Y., WIEDERHOLD, G., AND SAGALOWICZ, D. Estimating block accesses in database organizations: A closed noniterative formula. *Commun. ACM 26*, 11 (Nov. 1983), 940–944.
32. WONG, H. K. T., LIU, H., OLKEN, F., ROTEM, D., AND WONG, L. Bit transposed files. In *Proceedings of the 11th Conference on Very Large Data Bases* (Stockholm, Aug. 21–23, 1985). pp. 448–457.
33. YAO, S. B. Approximating block accesses in database organizations. *Commun. ACM 20*, 4 (Apr. 1977), 260–261.
34. ZIPF, G. *Human Behaviour and the Principle of Least Effort: An Introduction to Human Ecology.* Hafner Publications, 1949.

Received July 1986; revised February 1987; accepted June 1, 1987