# An Efficient Indexing Technique for Full-Text Database Systems

Justin Zobel
Department of Computer Science,
Royal Melbourne Institute of Technology,
GPO Box 2476V, Melbourne 3001, Australia.
jz@kbs.citri.edu.au

Alistair Moffat
Department of Computer Science
The University of Melbourne, Parkville 3052, Australia.
alistair@cs.mu.oz.au

Ron Sacks-Davis
Collaborative Information Technology Research Institute
723 Swanston St., Carlton 3053, Australia.
rsd@kbs.citri.edu.au

**Abstract:** Full-text database systems require an index to allow fast access to documents based on their content. We propose an inverted file indexing scheme based on compression. This scheme allows users to retrieve documents using words occurring in the documents, sequences of adjacent words, and statistical ranking techniques. The compression methods chosen ensure that the storage requirements are small and that dynamic update is straightforward. The only assumption that we make is that sufficient main memory is available to support an in-memory vocabulary; given this assumption, the method we describe requires at most one disc access per query term to identify answers to queries.

**Keywords:** Text support, storage management, indexing, compression, database performance

**Proceedings of the 18th VLDB Conference
Vancouver, British Columbia, Canada 1992**

## 1 Introduction

Full-text database systems can be used for storing and accessing document collections such as newspaper archives, office automation systems, and even libraries of books and articles. The needs of full-text databases are not well served by traditional database systems, since, instead of key indexing, full text requires facilities such as document ranking and indexing on text content.

In this paper we consider how best to implement the indexing component of a full-text database system. For a full-text database system, indexes should efficiently support three kinds of activity. First, given a boolean query on a set of words, the index should efficiently support retrieval of the documents, or more generally *records*, satisfying the query; in the context of text retrieval, *conjunctive queries* are particularly common. This requires a *record-level* index which indicates whether or not a record contains a word; indexes are *word-level* if word position is also stored. Second, it must be possible to efficiently insert new records; because of the archival nature of most full-text applications, deletion and change are less common, but should still be supported. Last, given an 'informal' query, it must be possible to statistically *rank* all of the records in the collection with respect to the query, so that the records most likely to be of interest to the user are retrieved first [SM83]; this strategy simplifies the problem of finding text about a topic. Basic ranking techniques require statistics about the words occurring in the collection, such as their frequency within the collection, so that words' importance can be estimated [SM83]. More sophisticated ranking techniques also use parameters such as the frequency of each word in each record and the length of each record.

Additional functionality may also be considered desirable. One addition is indexing on adjacency of words, allowing queries on phrases or *word sequences* as well as individual words. More generally, we might seek records in which two words lie within some specified distance of each other. Another possible extension is indexing on stems, substrings, patterns, and other partially specified terms.

We propose an inverted file scheme based on compression. We have already described a variety of fast algorithms for compressing sparse bitmaps; using these algorithms, record-level indexes can be represented in under 10% of the size of the source text [MZ92b]. Elsewhere we have argued that stored text should be compressed using a word-based model, in which each byte of text can typically be represented in about 2.2 bits [MZ92a]. Thus the compressed text of and a record-level index for a large database can be stored in approximately 40% of the space required by the source database. The compression of the text does not significantly affect retrieval time, as the decoding procedures are fast and the cost of decompression is partially offset by faster transmission from disc [ZM92a]. The only assumption that we make is that sufficient memory is available to support an in-memory vocabulary of the words used in the collection, together with some small amount of additional information associated with each word.

In this paper we show how, given the same assumptions, these methods for indexing text can be extended to efficiently support ranking and word-level indexing, which permits retrieval of word sequences. The large memories of current machines also allow in-memory storage of other tables that can be used to improve efficiency. Storage of vocabularies and other tables in-memory gives us better performance than traditional inverted file techniques [SM83].

Signature file methods have also been proposed for indexing text [CS88, Fal85a]. Signatures use hashing to form compacted representations of the data being indexed and therefore do not require a vocabulary to support boolean queries. For efficient query evaluation on large databases, signatures are usually transposed into bit slices [SDKR87, WLO$^+$85]. However, given sufficient memory to store the vocabulary, or a large component of it, many of the advantages of bit sliced signature file methods are lost. In particular, we show that bit sliced signature files require at least as many disc accesses to identify answers as do our inverted files. Furthermore, parameters such as the frequency of each word in each record cannot be stored with signature files, nor can they easily support word-level indexing.

In the following section we give an overview of the proposed method. In Section 3 we describe our scheme's memory requirements. We describe techniques for compressing the size of inverted file indexes in Section 4 and review previous work. In Section 5 we develop methods for supporting retrieval of word sequences. In Section 6 we outline a method for managing compressed indexes on secondary storage. In Section 7 we compare our methods to previous techniques. Conclusions are given in Section 8.

## 2 Inverted file text indexing

A general inverted file index consists of two parts: a set of *inverted file entries*, being lists of identifiers of the records containing each indexed word; and a *search structure* or *vocabulary* for identifying the location of the inverted file entry for each word (Figure 1). We assume that inverted file entries store ordinal record numbers rather than addresses, and so to map the resulting record identifiers to disc addresses there must also be an *address table* (or *disc mapping*). An entry in a record-level inverted file can be thought of as a representation of a bit vector in which the $i$th bit is set if the $i$th record contains the word being indexed.

For our indexing scheme we assume that there is sufficient main memory to hold the search structure. Given this assumption, the cost of finding candidate answers to boolean queries is at most one disc access for each word specified in the query, where a disc access involves a seek followed by retrieval of the one or more pages that contain the inverted file entry for that word. In a conjunctive query, fewer disc accesses may be needed, as retrieval of a series of inverted file entries may leave so few candidate answers that it is cheaper to fetch those records and check for *false matches* than to fetch the remaining inverted file entries. Once the candidates have been identified, either one or two disc accesses are required per candidate answer, depending on whether the address table is held in memory or on disc.

For ranking queries, as for boolean queries, at most one disc access per query term is required to identify answers. Basic ranking techniques require only the number of records containing each word to determine the likely relevance of each answer [SM83]; this frequency must be stored anyway for the index compression techniques. To improve ranking perfor-

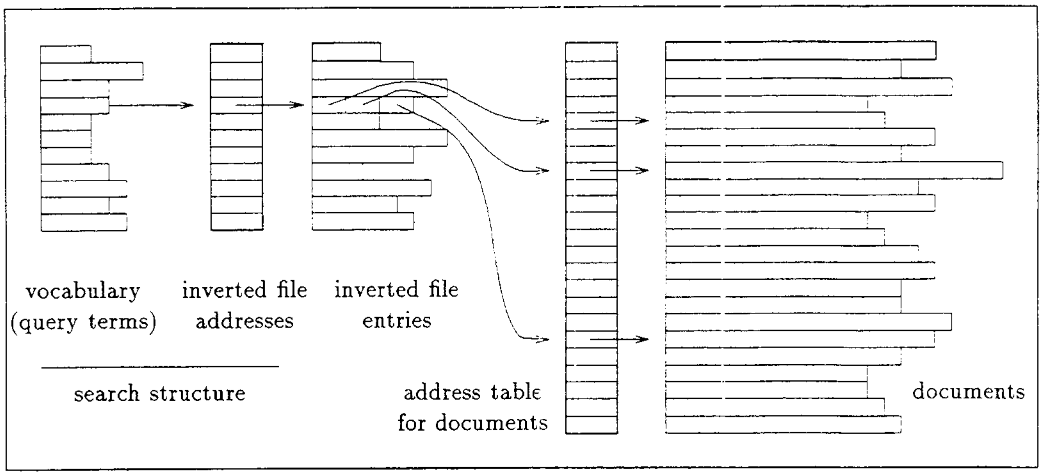

Figure 1: Inverted File Index Structure

mance (that is, recall and precision [SM83]), more information is needed. For example, better performance can be obtained if the frequency of each word in each record is known, and further improvement can be had by using the 'length' of each record; length might be a sum of weights of the words in the record or simply the number of words it contains. Croft & Savino estimate that the improvement in ranking performance in going from a basic scheme to more sophisticated schemes is about 10% to 20% [CS88].

The choice of ranking scheme will be affected by issues such as frequency of update. For example, inserting a new record means that each of the words in the record now occurs in one more record overall, thus changing the weight of each of these words. If record length was based on sums of weights, the length of every record containing any one of these words would have to be recomputed. In many applications it would be preferable to use a simpler measure of length, for which ranking performance may be worse but update is feasible.

## 3 Memory requirements

Our indexing scheme requires memory to hold the vocabulary of the indexed text collection. As well as the words themselves, the vocabulary can include pointers to index entries, frequency counts, and compression parameters. Memory is also used for buffers to hold data read from disc, and may be required for ancillary tables such as the address table and any data used for ranking.

Of these, the most important is the vocabulary. Fortunately, the size of the vocabulary of a text collection is usually small compared to the collection size. For example, the King James version of the Bible contains 13,777 distinct words—using case-sensitive comparison and distinguishing plurals, etc., from their roots—of 885,009 word occurrences. The largest of our text collections (described in the next section) contains 68,074 distinct words of 23,100,786 word occurrences; this vocabulary occupies 1.3 Mb including pointers to index entries, frequency counts, and a compression code. Harman & Candela describe an 806 Mb text collection that contains 243,470 distinct words [HC90]. If necessary, compression techniques can be used to reduce the memory requirements of the vocabulary [WBN91], and need not be computationally expensive [MZ92a], so precise choice of vocabulary representation will depend on the relative importance of fast search, space constraints, and fast insertion of new words.

If the search structure does not fit into memory, it could be partitioned, with an abridged vocabulary of common words held in memory and the remainder held on disc. This would still be effective because for large vocabularies many words only occur once or twice in the collection [Zip49], and so the cost of going to disc twice for rare words is offset by the fact that they dramatically reduce the set of candidate records. We believe, however, that the trend of growth in memory size will render partitioning unnecessary; for example, 16 Mb of memory should comfortably hold a vocabulary of 1,000,000 words as well as the other tables our indexing techniques require.

Other components may also be retained in main memory. For ranking purposes, the length of each record may be needed. Any straightforward represen-

| | *Manuals* | *GNUbib* | *Comact* |
|---|---|---|---|
| Text size (Mb) | 5.15 | 14.12 | 132.11 |
| Records | 2,496 | 64,344 | 261,829 |
| Distinct words | 27,554 | 70,866 | 68,074 |
| Word occurrences | 958,744 | 2,575,411 | 23,100,786 |
| Words per record (av.) | 384 | 40 | 88 |
| Distinct words per record (av.) | 169 | 36 | 51 |
| Distinct pairs per record (av.) | 332 | 39 | 73 |
| Inverted file size (Mb) | 0.45 | 4.38 | 30.51 |

Table 1: Sizes of Document Collections

tation of lengths will typically require one to four bytes per record, and in a large collection this could occupy more space than the vocabulary.

An address table will also be required. If main memory is scarce this would be implemented as a two level index and retrieval of each answer would require two disc accesses. However, single-access retrieval would be possible given as little as one bit of main memory per record, assuming suitable blocking of data on disc [MZ92b].

## 4 Compressing inverted files

In this section we consider methods for compressing inverted file entries. In all of the schemes we outline decompression is fast—about 50 Kb of compressed data can be decompressed in a second on a 25 MIP Sun SPARCserver 2—so that the decompression time is only a small overhead on the cost of the disc access for all but very frequent terms. Techniques for compressing inverted file entries, or equivalently bitmaps, have been described by many authors, including Fraenkel & Klein [FK85] and Bookstein & Klein [BK91]. Faloutsos described the application of similar techniques to the compression of sparse signatures [Fal85a, Fal85b].

Our presentation is based on that of Moffat & Zobel [MZ92b], who compare a variety of bitmap compression techniques. These techniques could be applied to any stream of bits, but, like most compression techniques, take advantage of sparsity and pattern in the values being compressed. To demonstrate the relative power of different compression schemes, we considered three text collections, *Manuals* (Unix manual pages), *GNUbib* (bibliographic citations), and *Comact* (acts of parliament). The parameters of these collections are shown in Table 1. The inverted file size is of an uncompressed record-level inverted file, assuming a binary code of $\lceil \log_2 N \rceil$ bits per record identifier, where $N$ is the number of records in the collection.

Rather than compressing the series of record numbers in an inverted file entry, we compress their *run-length encoding*, the series of differences between successive numbers [GV75, McI82]. For example, the inverted file entry

$$4, 5, 9, 11, 12, 17, \ldots$$

has the run-length encoding

$$4, 1, 4, 2, 1, 5, \ldots$$

This does not in itself yield any compression, but does expose patterns that can be exploited for compression purposes.

| | Coding method | |
|---|---|---|
| $x$ | $\gamma$ | $\delta$ |
| 1 | 1, | 1, |
| 2 | 01,0 | 010,0 |
| 3 | 01,1 | 010,1 |
| 4 | 001,00 | 011,00 |
| 5 | 001,01 | 011,01 |
| 6 | 001,10 | 011,10 |
| 7 | 001,11 | 011,11 |
| 8 | 0001,000 | 00100,000 |

Table 2: Examples of Codes

A simple run-length compression method is to use the codes for integers described by Elias [Eli75]. His $\gamma$ code represents integer $x$ as $\lfloor \log_2 x \rfloor + 1$ in unary (that is, $\lfloor \log_2 x \rfloor$ 0-bits followed by a 1-bit) followed by $x - 2^{\lfloor \log_2 x \rfloor}$ in binary (that is, $x$ less its most significant bit); the $\delta$ code uses $\gamma$ to code $\lfloor \log_2 x \rfloor + 1$, followed by the same suffix. Some sample values of codes $\gamma$ and $\delta$ are shown in Table 2; commas have been used to separate the suffixes and prefixes. The $\delta$ code is

| | *Manuals* | *GNUbib* | *Comact* |
|---|---|---|---|
| Binary (%) | 8.8 | 31.0 | 23.1 |
| $\gamma$ (%) | 5.1 | 12.3 | 6.5 |
| $\delta$ (%) | 4.8 | 10.7 | 6.2 |
| $V_T$ (%) | 4.5 | 9.9 | 6.0 |

Table 3: Space Requirements for Inverted Files

longer than the $\gamma$ code for most values of $x$ smaller than 15, but thereafter $\delta$ is never worse.

The compression of inverted file entries that can be achieved with $\gamma$ and $\delta$ is shown in Table 3. Each number in this table is the size of an index as a percentage of the size of its source text. For example, a set of compressed inverted file entries for *Comact* occupying 20 Mb would be described as having a space requirement of $20/132.11 = 15.1\%$.

The $\gamma$ and $\delta$ codes are instances of a more general coding paradigm as follows [FK85]. Let $V$ be a (possibly infinite) vector of positive integers $v_i$, where $\sum v_i \geq N$. To code integer $x \geq 1$ relative to $V$ we find $k$ such that

$$\sum_{j=1}^{k-1} v_j < x \leq \sum_{j=1}^{k} v_j$$

and code $k$ in some representation followed by the difference

$$d = x - \sum_{j=1}^{k-1} v_j - 1$$

in binary, using either $\lfloor \log_2 v_k \rfloor$ bits if $d < 2^{\lceil \log_2 v_k \rceil} - v_k$ or $\lceil \log_2 v_k \rceil$ bits otherwise. In this framework the $\gamma$ code is an encoding relative to the vector $(1, 2, 4, 8, 16, \ldots)$, with $k$ coded in unary.

Consider another example. Suppose that the coding vector is (for some reason) chosen to be $(9, 27, 81, \ldots)$. Then if $k$ is coded in unary, the values 1 through to 7 would have codes 1,000 through to 1,110, with 8 and 9 as 1,1110 and 1,1111 respectively, where again the comma is purely indicative. Similarly, run-lengths of 10 through to 36 would be assigned codes with a 01 prefix and either a 4-bit or a 5-bit suffix: 0000 for 10 through to 0100 for 14, then 01010 for 15 through to 11111 for 36.

The effectiveness of compression for an inverted file entry will vary with the choice of vector. One scheme, due to Teuhola [Teu78], is to use the vector

$$V_T = (b, 2b, 4b, 8b, 16b, \ldots)\,.$$

An appropriate choice of $b$ is the median run-length in the entry [MZ92b], again with $k$ coded in unary. This scheme gives good compression because it exploits skewness in entries; records containing text on a similar topic will often be clustered, since an effective way to store long documents is to break them into a series of adjacent records [ZTSD91]. For each inverted file entry, $b$ must be stored, either with the entry or in the vocabulary: $\gamma$ can be used to encode $N/b$, which will generally be smaller than $b$ itself. Compression (including this overhead) using $V_T$ is shown in Table 3.

Better compression can be achieved with the LLRUN technique, in which $k$ is coded with a Huffman technique based on observed values of $k$ for all inverted file entries [FK85, MZ92b]. However, LLRUN is effectively two pass; the entries must be created, then compressed in a batch. For the other schemes, entries can be efficiently created and compressed on the fly [Mof92], and it is not necessary to periodically rebuild a set of entries. (We discuss management of variable length entries in Section 6.) It follows that LLRUN is not likely to be suitable if updates are frequent. In general, any scheme in which parameters must be computed for effective compression may be inefficient with regard to update. In this respect, simple schemes such as the $\delta$ code may be preferable, as they achieve good compression with a minimum of processing overhead. All of the schemes described in this section allow entries to be individually modified.

On average, entries can be decompressed much more quickly than they can be retrieved. For example, the average *Comact* entry is 122 bytes, which can be decompressed in about 2 milliseconds. However, the longest entry is about 32 Kb, and we are investigating auxiliary structures to allow fast random access into compressed inverted file entries. We believe that it will be possible to provide fast AND operations on long inverted file entries whenever the set of candidate records is small.

None of our compression schemes use arithmetic coding or adaptive modelling, neither of which are effective in this application [BWC89]. Arithmetic coding requires significant computational resources, whereas our schemes require only a few machine instructions to decode each bit. Adaptive modelling requires long runs of data to be effective.

## 5 Extended indexing schemes

In this section we consider how to extend indexes to provide greater functionality, and the effect this has on space requirements. We first consider improving ranking, then indexing on word sequences.

| | *Manuals* | *GNUbib* | *Comact* |
|---|---|---|---|
| Frequency counts | | | |
| $\gamma$ (%) | 1.8 | 2.3 | 2.2 |
| Locations within record | | | |
| $\delta$ (%) | 24.1 | 17.7 | 17.7 |
| Global Bernoulli (%) | 20.3 | 13.5 | 14.0 |
| Local Bernoulli (%) | 18.5 | 13.0 | 13.3 |
| Total index size | | | |
| $V_T + \gamma +$ global Bernoulli (%) | 26.5 | 25.6 | 22.2 |
| Word-level index using $V_T$ (%) | 23.9 | 23.0 | 20.2 |
| $V_T + \gamma +$ signature (estimated) (%) | 21.6 | 28.7 | 22.0 |

Table 4: Space Requirements for Extended Indexes

## 5.1 Storing word frequency

Ranking is more effective if, in addition to each word's overall frequency, its frequency in each record is also known [SM83]. A simple way to achieve this is to interleave frequency counts with record identifiers in the inverted file entries. Since most frequencies are small, the $\gamma$ code is suitable for this task, although, if the great majority of frequency counts are 1 (as is true for our collections), still yields far from optimal compression [MZ92b]. In the first section of Table 4 we show, as a percentage of the size of the source text, the additional space required by frequency counts coded using $\gamma$.

## 5.2 Indexing word sequences

In this and the following two sections we consider three methods for indexing word sequences in association with an inverted file index. The method considered in this section is to extend the index to identify the locations where the words occur in each record. This also allows queries such as retrieval of records in which certain words are (say) less than ten words apart.

For each word, the sequence in the word's index entry for each record would be as follows:

| record number | frequency | positions ... |
|---|---|---|

A position number of $k$ indicates that the word is the $k$th in the record. Record number can be represented as a run length (distance from the previous record number) and compressed using $V_T$, as described in Section 4; frequency in record can be represented with $\gamma$ values; and the positions can also be represented as a sequence of run lengths. Since position numbers will generally be small—depending on the length in words of a typical record—a parameterless code, such as $\delta$, is one possibility for representing these run lengths. The first row of the second section of Table 4 shows, as a percentage of the size of the source text, the extra cost associated with the use of $\delta$ to code word locations.

Better compression is achieved with a Bernoulli model. This model assumes that each position number is independently likely to occur with probability $r$, so that a difference of $k$ between two position numbers will appear with probability $(1-r)^{k-1}r$, which is the geometric distribution. These probabilities can then be used implicitly to generate an infinite Huffman code [Fal85b, GV75, McI82, MZ92a, MZ92b] for coding run lengths. To calculate the code, we need an estimate for $r$, and take $r \approx p/N$, where $p$ is the number of occurrences of the word and $N$ is the total number of word occurrences. We then have a further choice: we can use localised values, where $p$ is the number of occurrences of the word in the record and $N$ is the number of words in the record; or we can take a global approach, where $p$ is the number of occurrences of the word in the database, and $N$ is the size in words of the database. The second section of Table 4 shows compression results for both of these variants. In the case of the local model, the compression rates listed include the small overhead cost of storing, for each document, its length in words; for the global model, the rates include the cost of storing the frequency of the word. In both cases the values were compressed using a suitable code.

The localised model gives slightly better compression than the global model, but requires that local values for $p$ and $N$ be known. Knowledge of the local value of $p$ is required in any case, to determine how many codes must be decoded. Knowledge of the local value of $N$ requires that an extra table, storing the length in words of each record, be held in memory. (The extra frequency value required by the global model can be stored on disc with the inverted file entry.) Although small in terms of the overall database, this table would require substantial amounts of main memory, and even compressed would require 200–300 Kb for the largest of our test databases. Unless the table of lengths is available anyway, to support ranking for example, we prefer the use of the global model for representing word locations. Bookstein, Klein, & Raita [BKR92] have also recently described the use of a Bernoulli model for storing positional information about words.

The third section of Table 4 shows the total cost of a record-level inverted index containing both word frequency and word locations. It is worth noting that a large fraction of this index is consumed by relatively few words. In our experiments we did not remove any stop words, and retained all case information, so that every sequence of alphanumeric characters was indexed. When no positional information is being recorded, case folding or the removal of stop words would achieve only small savings, since record-level inverted file entries for common words are represented very compactly in our coding methods. However, when positional information is added the inverted file entries for common words become dramatically larger. For example, on *Comact* the inverted file entry for 'the' grew from 32 Kb to 1 Mb, and the 5 most common words ('the', 'of', 'to', 'a', 'in') accounted for more than 3 Mbyte, 10% of the inverted file. In this case substantial additional savings would be gained by *not* recording positional information for a few common words, retaining only the record-level inverted file entry. Any positional or word sequence queries involving these words would then have to be evaluated with a relatively costly post-retrieval scan to eliminate the (many) false matches. It could be argued, however, that queries on such common words would be rare and that the space saving warranted the small risk of costly queries.

## 5.3 Word-level indexing

The second method of providing positional information is to store it directly within a word-level index. If a table of cumulative record lengths is held in memory, so that word positions (within the entire database) can be turned into record numbers, processing of both conjunctive queries and word sequence queries is possible [WBN91]. This would, however, make the simple queries slower, since the auxiliary record length table must searched for every word appearance to determine if two words appear in the same record, and has the disadvantage of requiring non-trivial amounts of main memory for the table of cumulative record lengths. The second row in the third section of Table 4 shows the cost of a complete word-level index compressed with $V_T$. There is a gain in compression, but in most cases this would be insufficient to warrant expenditure of several hundred kilobytes of main memory and additional time during query processing.

## 5.4 Use of a signature file for word pairs

The third method for supporting indexing on word sequences is to have a bit sliced signature file of adjacent word pairs only, to be accessed after the inverted file entries have been used to generate a list of candidate records. Single-level bit sliced signature files contain one fixed-length slice for each bit position; the length of each slice is the number of records being indexed [SDKR87]. To answer queries, a subset of the slices (corresponding to bits set by the query term) is retrieved and AND'ed together to identify candidate records. Bits in the same slice can be set by hashing different words. so there can be false matches which must be filtered out.

The effectiveness of a signature file that indexes word pairs will depend on the likely number of false matches. Without collecting real word sequence queries it is difficult to identify pairs likely to occur in practice, but using a heuristic we estimated the false match rate. We used a dictionary to identify the words in *Comact* that were nouns or adjectives, eliminating words that also had other senses; for example, the word 'are' is a verb as well as a noun (one hundredth of a hectare). We then found all pairs of nouns and adjectives and for each pair counted the number of records containing the pair (true matches) and the number of records containing both words but not containing the pair (false matches). On average, only 15% of matches were true matches. The proportion of true matches to false matches in real queries is unknown, but real pairs could include words of ambiguous sense such as 'will' and 'power', for which false matches are more likely.

We therefore aim to substantially reduce the number of false matches. Consider a signature file scheme in which hashing each pair sets two bits, with the signature width chosen so that on average at most 25% of the bits in each slice are on. For example, this could be achieved by choosing a signature width (in bits) 8 times the average number of distinct word pairs per record in the collection. To answer a word pair query, retrieval of two bit slices will, on average, eliminate $15/16 = 94\%$ of false matches. For our estimate of numbers of true matches, of the remaining candidates approximately 75% would be true matches. This is consistent with results of Sacks-Davis, Kent, & Ramamohanarao, who randomly selected pairs remaining when common words are eliminated [SDKR87]. Note that this method does not support queries where, for example, two words are required to be less than ten words apart within the record.

An estimate of the total storage cost for this alternative is given in the third section of Table 4, where it is assumed that: a record-level index is used; the $\gamma$ code is used for word frequency; and an auxiliary bit sliced signature file is used to process queries involving word sequences. The space required by this alternative is similar to that required by the extended inverted file indexes, but less functionality is provided, word sequence queries would require additional disc accesses, and post-retrieval scans are required.

## 6 Storage of variable-length entries

There are two difficulties in managing inverted file entries on disc. The first is that entries vary in length. The second is that, in a dynamic environment, entries can grow as new records are inserted. These problems are very similar to the more general problem of storage of variable-length modifiable records. However, compressed inverted file entries are on average relatively small; for our test collections, the largest average entry length was 468 bytes, for the word-level *Comact* index. Also, entries can be stored in any order.

We investigated a simple entry management scheme in which entries were stored in large, fixed-length blocks on disc. The details of this scheme are beyond the scope of this paper, but its broad outlines are as follows.

By using large blocks, many entries can be kept in each block, thus reducing space wastage. Each block contains a set of entries and a table indicating where each entry resides in the block. To retrieve an entry, the block containing the entry is fetched; this saves space in the vocabulary, as instead of a full address only a block number needs to be stored.

During insertion, deletion, and extension or contraction of an entry, the free space in each block can be governed by a fixed tolerance. If the free space in a block is less than the tolerance, no action need be taken. If the free space is greater than the tolerance, entries can be migrated between blocks, or a description of the block can be added to an in-memory 'free list', of blocks with space for new or modified entries. The small number of entries that are longer than a block must be managed separately.

In simulations with block sizes of 32 Kb and 64 Kb, record-level *Comact* index entries, and a tolerance of 2%, the free list never contained more than a few entries. Including the space required for the block tables, the space utilisation was better than 90%. These simulations also indicated that the average number of disc accesses required for an update was about one: updates require up to two accesses, but a series of insertions can be written to the last block, which can be held in memory for efficiency.

One overhead of using large blocks is retrieval time. With current disc performance, it takes roughly 50% longer to fetch a block of 64 Kb than a block of 1 Kb.[1] To improve retrieval time blocks must be shorter, which will lead to poorer space utilisation. This will often be an acceptable tradeoff since indexes are small compared to the source data. Another overhead is moving entries within a block to accommodate enlarged entries. However, moving half of a block's contents, as required for an average update in place, will take considerably less time than a single disc access for reasonable block sizes. For example, on the SPARCserver 2, 32 Kb can be moved in about 8 milliseconds, compared to about 20 milliseconds for a random disc access.

Although individual entries can be updated quickly with this scheme, insertion of a new record can still be expensive, since the entry for each of the record's words must be updated. For this reason, it may be desirable to batch insertions of new records: because new records will often have words in common, the cost per record will be significantly reduced.

## 7 Other methods

Harman & Candela [HC90] have recently described an inverted file implementation for full-text databases. Like our method, their scheme uses an in-memory vocabulary that includes frequency information and, on disc, each inverted file entry is sorted and contains record numbers and the frequency of each word in each record. This scheme has been designed to support ranking but could also be used for boolean queries.

Their scheme achieves good response times, but is not space efficient. On the collections for which sizes of indexes are given, each entry uses about 44 bits to store each record number/frequency pair, compared to 6.5 bits under our scheme for *Comact*, using $V_T$ with $\gamma$ frequencies. The (record-level) index for their largest collection appears relatively small—112 Mb for a 806 Mb collection—but this is because the average frequency of a word in a document is 6, compared to 1.6 for *Comact*, and because the average document in their collection contains 3,264 words, compared with

[1] We note, however, that typical figures for disc retrieval are not very informative. For example, for the three kinds of disc drives attached to our SPARCserver 2, the overheads of retrieving 64 Kb compared to retrieving 1 Kb were 30%, 45%, and 400%. Factors involved include seek, latency, caching and prefetch strategies, and disc channel speed.

88 for *Comact*.

Record-level inverted file indexes provide similar functionality to bit sliced signature files and it is instructive to compare them.

Inverted file indexes with in-memory search structures require fewer disc accesses to answer a conjunctive query than do bit sliced signature files. This can be seen from the following inductive argument. Initially, all records are candidate answers to the query. For each word in a query, there can be bits set in the word's bit slices that are not set in the word's bit vector (that is, inverted file entry), but the converse does not hold; so the word's bit vector is never denser than any of its bit slices. Thus, for any bit slice that can be selected to AND with the list of candidate records in a signature file index, a bit vector that is no less discriminating can be selected in the corresponding inverted file index. For a query involving $q$ words the selection of $q$ inverted file entries is sufficient to guarantee that no false matches remain in the inverted file case; but after $q$ bit slices have been processed in the signature file case there can still be candidate records that are false matches. These false matches can only be resolved after further disc accesses, either to retrieve more bit slices or to retrieve text. If, in the bit sliced case, $p$ slices are retrieved before false match checking begins, where $p < q$, then in the inverted file case false match checking should also be commenced after $p$ inverted file entries have been merged, and, provided that inverted file entries have been selected for merging in order of increasing length, the number of records that must be retrieved in the inverted file case cannot be greater than the number of records accessed in the signature file case.

Signature files can be augmented with an in-memory vocabulary of common words, and on-disc compressed bitvectors of the records containing the words [SDKR87]. From this perspective, signature files and record-level inverted files can be seen as extremes of a spectrum. Traversing the spectrum, we find at the signature file end the ability to handle an infinite vocabulary, and at the other end the ability to answer queries with a minimum number of disc accesses.

In practice, the relative performance of signature files and inverted files will also be affected by factors such as the lengths of bit slices and entries. Decoding a long inverted file entry can take much longer than a disc access, so in our scheme it is important that inverted file entries be retrieved in order of increasing length, and that the further AND'ing of entries cease as soon as the number of candidate records falls below some minimum threshold. If this is done it is necessary to check for false matches, exactly as if a signature file was used.

It is not clear which system these factors favour. Bit slices are of fixed length, with (typically) one bit per 4 to 32 records if blocking is used to reduce their size [SDKR87], so *Comact* entries can be as short as 1 Kb. The cost of composing two bit slices is fixed. Record-level inverted file entries vary in length, from a few bytes to about one bit for each record in the database, with an average of 122 bytes for *Comact*. The search structure can be used to order the selection of entries so that more discriminating entries are retrieved first, but nonetheless processing costs are variable: for words with short entries the cost of merging will be less than the cost of composing two bit slices, but for long, non-discriminating entries the cost will be greater.

As Croft & Savino have shown, inverted files support ranking more efficiently than do signature files [CS88]. Ranked queries behave like disjunctions of the words in the query [SFW83], and moreover, if ranking is to be effective, require a vocabulary that holds information about the relative importance of words. Inverted files are faster for disjunctive queries, and because there is an in-memory vocabulary it is possible to support queries on stems, patterns, and substrings [ZM92b]. They are also better at indexing collections in which the lengths of records is variable.

On the other hand, an inverted file approach requires that the text be inverted to generate the file, a process not necessary during signature construction. For example, Harman & Candela [HC90] report that inversion of a 50 Mb database using a disc-bound technique required over 10 hours. However, the inversion of large texts is a further application of the compressed inverted file representations that we advocate, and we have described an in-memory inversion technique that allowed inversion of *Comact* (132 Mb) in 45 minutes on the 25 MIP SPARCserver 2 [Mof92]. An area of current investigation is the extension of this technique to arbitrarily large databases.

Finally, even our word-level indexing schemes use less space than is typically required by conventional bit sliced signature files—quoted at 30% to 40% of the size of the source data [KSDR90, SDKR87]—and much less space than the 50% to 300% for inverted files quoted by Hasking [Has91].

## 8 Conclusion

We have described an indexing method based on compression for use in full-text retrieval databases. The assumptions we make are that we are working only with text and that the vocabulary is sufficiently restricted that it can be stored in main memory.

Given these assumptions, the indexing scheme provides fast response to boolean queries, and can be extended to support word sequence queries and ranking techniques. These indexes can be dynamically maintained, and it is not necessary to periodically rebuild them. At most one disc access per query term is required to identify answers, and we have shown that this need never be more than is required by a bit sliced signature file.

We have also shown that our techniques make efficient use of space. If conjunctive boolean queries or basic ranking are the only access to be supported, a typical compressed inverted file index will require less than 10% of the space used by the source text. If better ranking or word sequence queries are to be supported, the index will typically require less than 25% of the space used by the source text. These sizes compare favourably with other implementations of inverted files and with bit-sliced signature files.

Our method has a number of drawbacks. Like other text indexing methods, insertion of new records is complex and is best handled by batching, and database creation can be expensive. Also, there is some possibility of a bottleneck during inverted file entry decoding if long entries must be processed to obtain a small number of answers to some query. We are currently investigating possible solutions to all of these problems, and, despite these drawbacks, feel that the space-efficiency of the method and its other advantages allow it to compete more than favourably with established techniques.

## Acknowledgements

We would like to thank Lachlan Andrew for implementing several of the tests described in this paper, and Alan Kent for a great deal of helpful advice and assistance. Thanks are also due to Tim Bell and Ian Witten: it was their comments that initiated this investigation. This work was supported by the Australian Research Council.

## References

[BK91] A. Bookstein and S.T. Klein. Generative models for bitmap sets with compression applications. In *Proc. SIGIR Conf. on Inf. Retrieval*, pages 63–71, Chicago, 1991.

[BKR92] A. Bookstein, S.T. Klein, and T. Raita. Model based concordance compression. In J.A. Storer and M. Cohn, editors, *Proc. IEEE Data Compression Conference*, pages 82–91, Snowbird, Utah, March 1992.

[BWC89] T.C. Bell, I.H. Witten, and J.G. Cleary. Modelling for text compression. *ACM Computing Surveys*, 21(4):557–592, 1989.

[CS88] W.B. Croft and P. Savino. Implementing ranking strategies using text signatures. *ACM Trans. on Office Inf. Sys.*, 6(1):42–62, 1988.

[Eli75] P. Elias. Universal codeword sets and representations of the integers. *IEEE Trans. on Inf. Theory*, IT-21:194–203, 1975.

[Fal85a] C. Faloutsos. Access methods for text. *ACM Computing Surveys*, 17(1):49–74, 1985.

[Fal85b] C. Faloutsos. Signature files: Design and performance comparison of some signature extraction methods. In *Proc. ACM-SIGMOD*, pages 63–82, Austin, Texas, 1985.

[FK85] A.S. Fraenkel and S.T. Klein. Novel compression of sparse bit-strings—Preliminary report. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words, Volume 12*, NATO ASI Series F, pages 169–183, Berlin, 1985. Springer-Verlag.

[GV75] R.G. Gallager and D.C. Van Voorhis. Optimal source codes for geometrically distributed alphabets. *IEEE Trans. on Inf. Theory*, IT-21(2):228–230, 1975.

[Has91] R.L. Hasking. Special purpose processors for text retrieval. *Data Engineering*, 4(1):16–29, 1991.

[HC90] D. Harman and G. Candela. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. *Journ. American Society for Inf. Science*, 41(8):581–589, 1990.

[KSDR90] A.J. Kent, R. Sacks-Davis, and K. Ramamohanarao. A signature file scheme based on multiple organisations for indexing very large text databases. *Journ. American Society for Inf. Science*, 41(7):508–534, 1990.

[McI82] M.D. McIlroy. Development of a spelling list. *IEEE Trans. on Communications*, COM-30(1):91–99, 1982.

[Mof92] A. Moffat. Economical inversion of large text files. *Computing Systems*, 5(2), June 1992. To appear.

[MZ92a] A. Moffat and J. Zobel. Coding for compression in full-text retrieval systems. In J.A. Storer and M. Cohn, editors, *Proc. IEEE Data Compression Conference*, pages 72–81, Snowbird, Utah, March 1992.

[MZ92b] A. Moffat and J. Zobel. Parameterised compression for sparse bitmaps. In *Proc. SIGIR Conf. on Inf. Retrieval*, Copenhagen, Denmark, June 1992. To appear.

[SDKR87] R. Sacks-Davis, A.J. Kent, and K. Ramamohanarao. Multi-key access methods based on superimposed coding techniques. *ACM Trans. on Database Systems*, 12(4):655–696, 1987.

[SFW83] G. Salton, E.A. Fox, and H. Wu. Extended Boolean information retrieval. *Comm. ACM*, 26(11):1022–1036, 1983.

[SM83] G. Salton and M.J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, New York, 1983.

[Teu78] J. Teuhola. A compression method for clustered bit-vectors. *Inf. Proc. Letters*, 7(6):308–311, 1978.

[WBN91] I.H. Witten, T.C. Bell, and C. Nevill. Models for compression in full-text retrieval systems. In J.A. Storer and J.H. Reif, editors, *Proc. IEEE Data Compression Conference*, pages 23–32, Snowbird, Utah, April 1991.

[WLO+85] H.K.T. Wong, H. Liu, F. Olken, D. Rotem, and L. Wong. Bit transposed files. In *Proc. VLDB*, pages 448–457, Stockholm, August 1985.

[Zip49] G. Zipf. *Human Behaviour and the Principle of Least Effort: An Introduction to Human Ecology.* Hafner Publications, 1949.

[ZM92a] J. Zobel and A. Moffat. Adding compression to a full-text retrieval system. In *Proc. 15'th Australian Computer Science Conference*, pages 1077–1089, Hobart, Australia, January 1992.

[ZM92b] J. Zobel and A. Moffat. Pattern indexing for main memory lexicons. Technical Report 92/28, Collaborative Information Technology Research Institute, Department of Computer Science, Royal Melbourne Institute of Technology, Australia, May 1992.

[ZTSD91] J. Zobel, J.A. Thom, and R. Sacks-Davis. Efficiency of nested relational document database systems. In *Proc. VLDB*, pages 91–102, Barcelona, Spain, 1991.