Adding Compression to Block Addressing Inverted Indexes *

Gonzalo Navarro [‡], Edleno Silva de Moura[†], Marden Neubert[†], Nivio Ziviani[†] and Ricardo Baeza-Yates[‡]
[‡] *Department of Computer Science, Univ. of Chile, Chile.*
({gnavarro,rbaeza}@dcc.uchile.cl*).*
[†] *Department of Computer Science, Univ. Federal de Minas Gerais, Brazil.*
({edleno,marden,nivio}@dcc.ufmg.br*).*

Abstract. Inverted index compression, block addressing and sequential search on compressed text are three techniques that have been separately developed for efficient, low-overhead text retrieval. Modern text compression techniques can reduce the text to less than 30% of its size and allow searching it directly and faster than the uncompressed text. Inverted index compression obtains significant reduction of their original size at the same processing speed. Block addressing makes the inverted lists point to text blocks instead of exact positions and pay the reduction in space with some sequential text scanning.

In this work we combine the three ideas in a single scheme. We present a compressed inverted file that indexes compressed text and uses block addressing. We consider different techniques to compress the index and study their performance with respect to the block size. We compare the index against three separate techniques for varying block sizes, showing that our index is superior to each isolated approach. For instance, with just 7% of extra space overhead the index has to scan less than 0.5% of the text for exact searches and about 1% allowing one error in the matches.

Keywords: Text compression, inverted files, block addressing, text databases.

1. Introduction

The large sizes of today's text collections demand specialized indexing techniques to allow fast access to documents searched by the users. An *index* is a persistent data structure built on the text in advance to speed up query processing. A simple indexing structure for large text collections is the *inverted index* (Baeza-Yates and Ribeiro-Neto, 1999; Witten et al., 1999). Inverted indexes are the most popular indexing technique for large text databases storing natural language documents.

An inverted index is typically composed of a vector containing all distinct words of the text collection in lexicographical order (which is

* This work has been partially supported by SIAM/DCC/UFMG Project, grant MCT/FINEP/PRONEX 76.97.1016.00, AMYRI/CYTED Project, CONICYT grant 1990627 (Gonzalo Navarro and Ricardo Baeza-Yates), CAPES scholarship (Edleno Silva de Moura), Miner Technology Group scholarship (Marden Neubert), and CNPq grant 520916/94-8 (Nivio Ziviani).

called the *vocabulary*) and, for each word in the vocabulary, a list of all text positions in which that word occurs (which is called the *list of occurrences*). The search for a word in an inverted index first locates the word in the vocabulary and then retrieves its list of text positions. To search for a phrase or proximity pattern (where the words must appear consecutively or close to each other, respectively), each word is searched separately and the lists of occurrences are later intersected taking care of the consecutiveness or closeness of the word positions in the text. Other recent proposals are presented in (Williams et al., 1999).

The most important considerations to evaluate the efficiency of an indexing scheme are its construction, updating and querying times, and its space requirements. Both the time and space requirements are a function of the granularity of the index, which defines what is the unit of information represented. Three different types of inverted indexes can be identified by their granularity, going from the faster to process queries, but slower to build the index and more space demanding, to the slower to process queries, but faster to build the index and less space demanding.

The first one is called a "full inverted index". Adequately implemented, it is the fastest index to solve most queries. It uses the simplest scheme, where the index points to all the positions of all the words in the text. However, its construction time and the space requirement are higher. The occurrences take nearly 60% of the text size. This can be reduced to 35% by omitting the *stopwords* from the vocabulary (Araújo et al., 1997). Stopwords are articles, prepositions, and other words that carry no meaning and therefore do not appear in or that can be removed from user queries. Stopwords represent 40% to 50% of all the text words. However, 35% of extra space can still be a high space requirement for a large text collection, and therefore different techniques exist to reduce the space taken by the lists of occurrences.

The second type is known as "inverted file". In an inverted file the lists of occurrences do not point to the exact occurrences of the words but just to the documents where each word appears. This saves space because all the occurrences of the same word in the same document are referenced only once, and the pointers may be smaller because there are less documents than text positions. Normal space requirements of inverted files are around 25% of the text size. Single word queries are solved directly in inverted files, without access to the text. This is because if a word appears in a document the whole document is retrieved. However, phrase or proximity queries cannot be solved with the information that the index stores. Two words can be in the same document but they may or may not form a phrase or be close. For these queries we must search directly the text of those documents where all the relevant words appear.

The third type, called "block addressing index", goes one step further (Manber and Wu, 1994; Baeza-Yates and Navarro, 2000). It divides the text in blocks of fixed size (which may span many documents, be part of a document or overlap with document boundaries). The index stores only the blocks where each word appears. Since normally there are much less blocks than documents, the space occupied by the index is very small and can be chosen according to the user needs. On the other hand, almost any query must be solved using some sequential searching on the text because it is not known in which documents of the block the word appears. The index is used just as a device to filter out some blocks of the collection that cannot contain a match.

This last index scheme was first proposed in Glimpse (Manber and Wu, 1994), which is a widely known system that uses block addressing indexes. One interesting use for Glimpse is to transparently index all the files of a user. The index takes little space and is kept updated by periodic rebuilds, and it allows finding, at any time, the user files containing a given pattern. Glimpse is also used to index Web sites, providing fast search on their Web pages with a low overhead index technique.

An orthogonal technique to reduce the space requirements of inverted indexes is compression. The text and/or the index can be compressed to reduce space requirements. The key idea to reduce the size of inverted indexes is that the list of occurrences of each word is in increasing order, and therefore the gaps between consecutive positions can be stored instead of the absolute values. Then, compression techniques for small integers can be used. As the gaps are smaller for longer lists, longer lists can be compressed better. Recent work has shown that inverted files can be reduced up to 10% of their original size without degrading the performance, and even the performance may improve because of reduced I/O (Witten et al., 1999).

Text compression seems difficult to combine with an inverted index because of the need to access the text at random positions (for presentation purposes) and to sequentially search parts of the text (which is required in inverted files and block addressing indexes). These needs traditionally demanded uncompressing the text from the beginning until reaching the desired part. Recent text compression techniques (Bentley et al., 1986; Moffat, 1989), however, not only allow reducing the text to 25% to 30% of its original size but also allows direct searching on the compressed text without decompressing (Turpin and Moffat, 1997; Moura et al., 2000), much faster than the same search done on the uncompressed text. Moreover, those compression techniques are based on Huffman coding where the symbol table is the vocabulary of the text. This makes them couple well with an inverted index.

Inverted file compression, block addressing and sequential search on compressed text have not been combined up to now in a single scheme, and this is precisely what we do in this work. We present a compressed inverted index that indexes compressed text and uses block addressing. We study specialized techniques to compress a block addressing index and study their performance with respect to the block size. The index is built by using an in memory compression scheme that improves the performance by reducing the I/O cost during the inversion (see also (Hawking, 1997)).

If we use our scheme to index all the user files (like Glimpse), the result would be an improvement both in the indexing and searching times. Furthermore, the overall space used by the compressed text plus the compressed index take nearly one third of the original uncompressed files without index. To make this arrangement transparent to the user, the system can be implemented as a compressed and indexed file system. A layer between the applications and the file system takes care of compressing and uncompressing the texts upon write and read operations, respectively, so that editors, Web servers and other applications work transparently with the files (as *Doublespace* in DOS or Windows). At the same time, the system keeps a small index that allows finding at any time the files containing a given user query.

We show that, for instance, 1 Gb of text can be indexed with only 7% of overhead and search one word patterns in 1.6 seconds (traversing less than 0.5% of the text) and phrase patterns in less than 0.3 seconds (traversing less than 0.1% of the text).

This work is organized as follows. Section 2 presents the compression technique based on Huffman coding on words. Section 3 explains how to search efficiently on compressed text. Section 4 presents the block addressing scheme. Section 5 presents techniques to compress the occurrences of an inverted index, and our proposals specialized for compressing block addressing indexes. Section 6 shows the complete scheme combining block addressing, index compression and text compression. Section 7 presents some experimental results between the combined approach and each isolated technique. Finally, Section 8 presents conclusions and future work directions.

2. Word-Based Byte-Oriented Huffman Compression

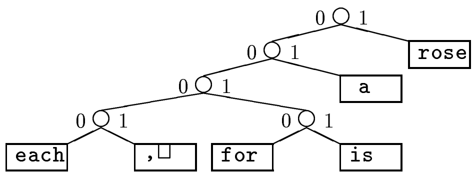
For large natural language texts used in an Information Retrieval (IR) context, the most effective compression technique is word-based Huffman coding (Huffman, 1952; Bentley et al., 1986; Moffat, 1989). The idea of Huffman coding is to assign a unique variable-length bit encoding to each different word of the text, and compression is achieved by assigning shorter codes to words with higher frequencies. Compression proceeds in two passes over the text. The encoder makes a first pass over the text to obtain the frequency of each different text word and performs the actual compression in a second pass.

Traditional implementations of the Huffman method are character-based, i.e., adopt the characters as the symbols in the alphabet. A successful idea towards merging the requirements of compression algorithms and the needs of IR systems is to consider that the symbols to be compressed are words and not characters (Bentley et al., 1986; Moffat, 1989). Words are the atoms on which most IR systems are built. Taking words as symbols means that the table of symbols in the compressor is exactly the vocabulary of the text, allowing a natural integration between an inverted file and a word-based Huffman compression method[1].

An important consideration is the size of the text vocabulary. An empirical law widely accepted in IR is Heaps' Law (Heaps, 1978), which states that the vocabulary of a text of n words is of size $V = O(n^\beta)$, where $0 < \beta < 1$ depends on the text. As shown in (Araújo et al., 1997), β is between 0.4 and 0.6 in practice, so the vocabulary needs space proportional to the square root of the text size. Hence, for large texts the overhead of storing the vocabulary is minimal. Another useful law related to the vocabulary is the Zipf's Law (Zipf, 1949), which states that the frequency in the text of the i-th most frequent word is $1/i^\theta$ times that of the most frequent word, where $\theta \geq 1$ is a constant that depends on the text. That is, in a text of n words the i-th most frequent word appears $n/(i^\theta H)$ times, where H is a constant that makes the frequencies to add up to n.

A natural language text is composed of words and of separators. An efficient way to deal with words and separators is to use a method called *spaceless words* (Moura et al., 2000). If a word is followed by a space, just the word is encoded. If not, the word and then the separator are encoded. At decoding time, it is assumed that a space follows each word, except if the next symbol corresponds to a separator. Figure 1 presents an example of compression using Huffman coding for the spaceless words method. The set of symbols in this case is {`"a"`, `"each"`, `"is"`, `"for"`, `"rose"`, `",␣"`}, whose frequencies are 2, 1, 1, 1, 3, 1, respectively.

[1] Some operations are inefficient, however, e.g. a file cannot be exported from the system in its current compressed form.



Original text: `for each rose, a rose is a rose`
Compressed text: `0010 0000 1 0001 01 1 0011 01 1`

Figure 1. Compression using Huffman coding for spaceless words.

The example also shows how the codes for the symbols are organized in a so-called *Huffman tree*. The Huffman method gives a tree that minimizes the length of the compressed file, but many trees would have achieved the same compression, e.g. left and right children can be exchanged at any node. The preferred choice for most applications is the *canonical tree*, where the right subtree is never taller than the left subtree, as is the case in Figure 1. Canonical trees allow more efficiency at decoding time with less memory requirement. Algorithms for building the Huffman tree from the symbol frequencies is described, for instance, in (Bell et al., 1990). They require $O(V \log V)$ worst case time, although due to the Zipf's distribution, the average time is linear in V.

Decompression is accomplished as follows. The stream of bits in the compressed file is traversed sequentially. The sequence of bits read is used to traverse the Huffman tree, starting at the root. Whenever a leaf node is reached, the corresponding word (which constitutes the decompressed symbol) is printed out and the tree traversal is restarted. Thus, according to the tree in Figure 1, the presence of the code `0010` in the compressed file leads to the decompressed symbol `"for"`.

The original method proposed by Huffman is mostly used as a binary code. In (Moura et al., 2000) the Huffman code assigned to each text word is a sequence of whole bytes and the Huffman tree nodes have degree 256 (called byte-oriented Huffman compression), instead of 2. For example, a possible Huffman code for the word `"rose"` could be the 3-byte code "`47 131 8`". Experimental results presented in (Moura et al., 2000) have shown that no significant degradation of the compression ratio is experienced by using bytes instead of bits when coding the words of a vocabulary. On the other hand, decompression and searching of byte Huffman code is faster than for binary Huffman code, because bit shifts and masking operations are not necessary.

One important consequence of using byte Huffman coding is the possibility of performing fast direct searching on compressed text. The search algorithm is explained in the next section. As seen in this work, this technique is not only useful to speed up sequential search, but it can also be used to improve indexed schemes that combine inverted files and sequential search.

3. Sequential Search on Compressed Text

We explain now how the compressed text can be searched (Moura et al., 2000). We start with a general technique that allows searching for very complex patterns and then consider possible speedups. The general technique will be presented from the simplest to the most complex scenario. The final setup allows a large number of variants, which forms a language originally defined for *Agrep* (Wu and Manber, 1992).

- Searching allowing errors (also called "approximate pattern matching"): given a query pattern and a number k, the system retrieves the occurrences of words which can be transformed into the query with up to k "errors". An error is the insertion, deletion or replacement of a character. For instance, searching `"color"` with $k = 1$ retrieves `"colour"` as well.

- Searching for classes of characters: each pattern position may match with a set of characters rather than with just one character. This allows some interesting queries:

  - range of characters (e.g. `t[a-z]xt`, where `[a-z]` means any letter between `a` and `z`);
  - arbitrary sets of characters (e.g. `t[aei]xt` meaning the words `taxt`, `text` and `tixt`);
  - complements (e.g. `t[∼ab]xt`, where `∼ab` means any single character except `a` or `b`; `t[∼a-d]xt`, where `∼a-d` means any single character except `a`, `b`, `c` or `d`);
  - arbitrary characters (e.g. `t·xt` means any character as the second character of the word);
  - case insensitive patterns (e.g. `Text` and `text` are considered as the same word).

- Searching for regular expressions (exactly or allowing errors). Some examples are:

  - unions (e.g. `t(e|ai)xt` means the words `text` and `taixt`);

  - arbitrary number of repetitions (e.g. `t(e|ai)*xt` means the words beginning with `t` followed by `e` or `ai` zero or more times followed by `xt`);
  - arbitrary number of characters in the middle of the pattern (e.g. `t.*xt`). It is customary to denote `.*` as `#`.

- Combining exact matching of some of their parts and approximate matching of other parts (e.g. `<te>xt`, with $k = 1$, meaning exact occurrence of `te` followed by an occurrence of `xt` with 1 error).

- Matching with nonuniform costs (e.g. the cost of insertions can be defined to be twice the cost of deletions).

3.1. A General Search Technique

Consider the search for a single word. The preprocessing consists of searching for it in the vocabulary and marking the corresponding entry, that is, a leaf of the Huffman tree. This search can be very efficient if binary search or hashing is used.

Next, we scan the compressed text, byte by byte, and at the same time traverse the Huffman tree downwards, as if we were decompressing the text. Each time we reach a leaf of the Huffman tree, we know that a word has been read, so we check if the leaf is marked, in which case we report an occurrence. Be the leaf marked or not, we return to the root of the Huffman tree and resume the text scanning.

If the pattern is not a simple word, we cannot perform a direct search in the vocabulary. In this case the preprocessing phase corresponds to a *sequential* vocabulary search to mark all the words that match the pattern. Specialized sequential algorithms are used to search allowing classes of characters, errors in the matches, regular expressions, multiple patterns and combinations. Since the vocabulary is small compared to the text size ($O(n^\beta)$ size, recall Section 2), a sequential search is feasible (some alternatives are considered in Section 4). The text scanning phase is exactly as before, the only difference being that more than one leaf of the Huffman tree may be marked.

Consider now the search for a phrase query. The phrase is a sequence of elements, each one a simple or complex pattern. Trying to extend the previous approach in a brute force fashion is not simple, because possible phrase occurrences may overlap with others, some words may match many phrase elements, etc.

If a phrase has ℓ elements, we set up a bit mask of ℓ bits for each word (i.e. leaf of the Huffman tree). The i-th bit of word x is set if x matches the i-th element of the phrase query. Then, each element

i of the phrase in turn is searched in the vocabulary and marks the i-th bit of the words it matches with. Note that some elements may be simple words searched with binary search or hashing and others may be complex patterns sequentially searched. Once this preprocessing has concluded, we scan the text as before. Each time we arrive to a leaf (i.e. word) we retrieve its bit mask, which indicates which phrase elements the word matches.

The search for phrases is organized using a nondeterministic automaton of $\ell + 1$ states. This automaton allows moving from state i to state $i + 1$ if the i-th element of the phrase is recognized. State zero is always active and occurrences are reported whenever state ℓ is activated. The automaton is nondeterministic because at a given moment many states may be active (i.e. many prefixes of the phrase may have matched the text).

Each time we reach a leaf of the Huffman tree, we send its bit mask to the automaton. An active state $i - 1$ will activate the state i only if the i-th bit of the mask is active. Therefore, the automaton makes one transition per word of the text. Figure 2 illustrates this phase for the pattern `"ro# rose is"` with $k = 1$ (i.e. allowing 1 error per word, where `"ro#"` means any word starting with `"ro"`). For instance, the word `"rose"` in the vocabulary matches the pattern in positions 1 and 2.

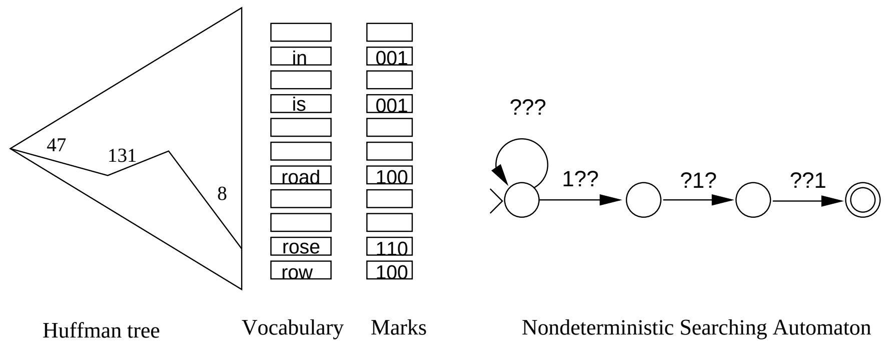


Figure 2. General searching scheme for the phrase `"ro# rose is"` allowing 1 error.

This scheme allows disregarding separators between the words of the phrase. That is, a phrase pattern can be found even if the separator between two words is a couple of spaces instead of one space. To achieve this, we ignore the leaves of the Huffman tree we arrive to if they are separators. In the same way, stopwords can be ignored in a phrase match. This is very difficult to do with typical sequential search algorithms on uncompressed text. On the other hand, we can avoid that a phrase spans more than one sentence, by taking into account separators that contain a period ("."). Those leaves of the Huffman tree will have a bit mask composed of ℓ zeros and therefore no phrase occurrence will contain them.

The remaining problem is how to implement this automaton efficiently. The algorithm of choice is Shift-Or (Baeza-Yates and Gonnet, 1992), which is able to simulate an automaton of up to w states (where w is the length in bits of the computer word) performing a constant number of operations per text character. In our case, it means that we can solve phrases of up to 32 or 64 words, depending on the machine, extremely fast. Longer phrases need the use of $\lceil \ell/w \rceil$ machine words for the simulation but the technique is the same.

The idea of the algorithm is to map each state of the automaton (except the first one) to a bit of the computer word. For each new text word (i.e. reached leaf of the suffix tree), each active state can activate the next one, which is simulated using a *shift* in the bit mask. Only those that match the current phrase element can actually pass, which is simulated by a bit-wise *and* operation with the bit mask found in the leaf of the Huffman tree. Therefore, with one *shift* and one *and* operation per text word the search state is updated (the original algorithm uses the reverse bits for efficiency, hence the name Shift-Or).

The search time is therefore $O(kn^{\beta}+n)$ in the worst case, where the first term comes from searching the vocabulary and the second from the text search ($O(1)$ operations per byte of the compressed text).

3.2. Faster Filters

The previous scheme is general and can cope with very complex searching. It is possible, however, to search faster. In particular, we are interested in not examining all the characters of the compressed text.

The simplest case to consider is the search for one single word. In this case, instead of the previous approach, we can simply find the word in the vocabulary, get its compressed code, and search for the code in the compressed text directly with any standard pattern matching algorithm. The resulting algorithm is as fast as the fastest search on uncompressed text, with the additional benefit of reduced I/O. In terms of elapsed time, the search is much faster if the text has to be read from disk.

One problem to be solved is the possibility of false matches: the compressed pattern can be present in the text just because it matches inside the concatenation of other compressed codes. This is solved by either using one bit of the bytes of the compressed codes to signal the beginning of each code, or by setting up synchronization points in the compressed text to which the codes are aligned. In the latter case each potential match must be followed by a verification starting in the last synchronization point.

We consider now more complex cases. In case of a single complex query, we find all the codes of the matching words and perform a multipattern search in the compressed text. In the case of a phrase, we choose one element as a representative, search it directly in the text, and verify the surrounding text of each match for complete phrase occurrences. The element to search can be chosen trying to make the search faster (e.g. that with longest code or least codes to search for).

The average time to search the text is improved with this scheme. If we search for a single pattern, then it is possible to obtain $O(n \log(c)/c)$ time, where c is the length in bytes of the compressed pattern. The complexity for a multipattern search has no closed expression (Baeza-Yates and Régnier, 1990).

4. Block Addressing

Block addressing is a technique to reduce the space requirements of an inverted index. It was first proposed in a system called *Glimpse* (Manber and Wu, 1994). The idea is that the text is logically divided in blocks, and the occurrences do not point to exact word positions but only to the blocks where the word appears. Space is saved because there are less blocks than text positions (and hence the pointers are shorter), and also because all the occurrences of a given word in a single text block are referenced only once. Figure 3 illustrates a block addressing index with r blocks of b words each (i.e. $n = rb$).

Searching in a block addressing index is similar to searching in a full inverted one. The pattern is searched in the vocabulary and a list of blocks where the pattern appears is retrieved. However, to obtain the exact pattern positions in the text, a sequential search over the qualifying blocks becomes necessary. The index is therefore used as a filter to avoid a sequential search over some blocks, while the others need to be checked. Hence, the reduction in space requirements is obtained at the expense of higher search costs.

At this point the reader may wonder what is the advantage of pointing to artificial blocks instead of pointing to documents (or files), this way following the natural divisions of the text collection. If we consider the case of simple queries (say, one word), where we are required to return only the list of matching documents, then pointing to documents is a very adequate choice. Moreover, as shown in (Baeza-Yates and Navarro, 2000), it may reduce space requirements with respect to using blocks of fixed size. Also, if we use blocks of fixed size and pack

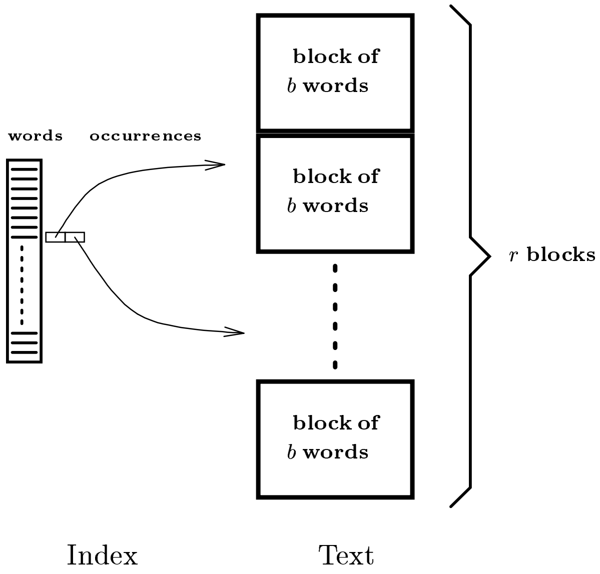


Figure 3. A block addressing index.

many short documents in a logical block, we will have to traverse the matching blocks (even for these simple queries) to determine which documents inside the block actually matched.

However, consider the case where we are required to deliver the exact positions which match a pattern. In this case we need to sequentially traverse the qualifying blocks or documents to find the exact positions. Moreover, in some important types of queries such as phrases or proximity queries, the index can only tell that two words appear in the same block, and we need to traverse it in order to determine if they form a phrase.

In this case, pointing to documents of different sizes is not a good idea because larger documents are searched with higher probability and searching them costs more. In fact, the expected cost of the search is directly related to the variance in the size of the pointed documents. This suggests that if the documents have different sizes it may be a good idea to (logically) partition large documents into blocks and to put small documents together, such that blocks of the same size are used.

Block addressing was analyzed in (Baeza-Yates and Navarro, 2000), where an important result is analytically proved and experimentally verified: a block addressing index may yield sublinear space overhead and at the same time sublinear query time. Traditional inverted indexes pointing to words or documents achieve only the second goal. It is shown in (Baeza-Yates and Navarro, 2000) that in order to obtain a space overhead of $\Theta(n^\gamma)$, it is necessary to set $b = \Theta(n^{(1-\gamma)/(1-\beta)})$, in which case the query time obtained is $O(n^\beta + n^{1-\beta+\alpha}b)$. In the formula, α is related to the query complexity: $O(n^\alpha)$ vocabulary words match the query, where $\alpha = 0$ for exact queries and $0 < \alpha < \beta$ for complex queries. The time complexity is sublinear for $\gamma > 1 - (1-\beta)(\beta - \alpha)$. In practice, $O(n^{0.85})$ space and query time can be obtained for exact queries.

Of course there is not an "optimal" block size but a space-time tradeoff related to b. The case $b = O(1)$ corresponds to full inverted indexes, whose space requirements are $O(n)$ and their time for a single word query is $O(n^\beta + n^{1-\beta+\alpha})$ (about $O(n^{0.5})$ for exact searching). The other extreme, $b = \Omega(n)$, means that just the vocabulary is stored, needing $O(n^\beta)$ (about $O(n^{0.5})$) space and $O(n)$ search time.

5. Index Compression

We show in this section how to compress inverted indexes in order to achieve significant space reduction and also allow fast access to the inverted lists. We also describe in this section a simple technique to improve the compression when using block addressing. The idea is to avoid storing the lists of words that appear in almost all the text blocks, therefore reducing the size of the index and the amount of processing in queries.

Comprehensive works showing how to compress inverted indexes can be found on the literature (Linoff and Stanfill, 1993; Moffat and Bell, 1995; Witten et al., 1999) and block addressing is just a type of inverted index. All these previous works are therefore useful here. The techniques used to compress inverted indexes can be classified in parameterized and non-parameterized. Parameterized techniques, such as *Golomb* codes (Golomb, 1966), produce different outputs depending on their parameters, so that one can adjust the coding scheme to the characteristics of the input to compress. Non-parameterized coding schemes do not need any information about the elements to be coded, so their output is fixed for each input. When using parameterized coding, the necessity of previous knowledge about the input requires two passes on the list of symbols to be coded, which can be a drawback if we are interested in good performance (in particular, the inability to generate the inverted list directly in compressed form may translate into higher I/O at indexing time). These two passes could be merged with the two passes that we need to obtain an approximation of the parameters, but decompression whould be slower anyway. Further, the best parameterized coding methods produce just slight better compression ratios when compared against the best non-parameterized methods. Our main focus when building block addressing indexes is to improve the performance. Therefore we use a non-parameterized scheme in this work.

Previous studies have already shown the best non-parameterized methods that can be used in inverted index compression (Moffat and Bell, 1995; Witten et al., 1999). For the sake of completeness we repeat here four important concepts: the gaps, Unary coding, Elias-γ coding, and Elias-δ coding.

Gaps: The block numbers are assigned incrementally during the parsing of the text, the pointers in each inverted list are in ascending order. Each non-initial pointer can then be substituted by the difference (or *gap*) from the previous number of the list. Since processing is usually done sequentially starting from the beginning of the list, the original block numbers can always be recomputed through sums of the gaps. The lists are now composed by smaller integers and we can obtain better compression using an encoding that represents shorter values in fewer bits.

Unary coding: A simple scheme codes an integer x in $(x-1)$ one-bits followed by a zero-bit and therefore is called *Unary code*. The unary codes for numbers 1 to 10 are shown in Table I.

Elias-γ coding: Elias (Elias, 1975) studied other variable-length encodings for integers. Elias-γ code represents an integer x by the concatenation of two parts, a unary code for $1 + \lfloor \log x \rfloor$ followed by a code of $\lfloor \log x \rfloor$ bits corresponding to $x - 2^{\lfloor \log x \rfloor}$ in binary. The total code length is thus $1+2\lfloor \log x \rfloor$. Some examples are presented in Table I.

Elias-δ coding: The other coding scheme introduced by Elias is the δ code, in which the prefix indicating the number of bits in the second part is coded in Elias-γ code rather than unary. The Elias-δ code for an integer x requires $1 + 2\lfloor \log \log 2x \rfloor + \lfloor \log x \rfloor$ bits. As Table I shows, for small values of x, Elias-γ codes are shorter than Elias-δ codes, but this situation is reversed as x grows. We will present experiments using both methods to compress the index in this work.

In particular, these techniques can be combined with methods that allow direct access to the list at periodic intervals (Witten et al., 1999), which permits to speed up phrase and conjunctive queries between very long and very short lists. We do not use these techniques in the present work.

5.1. Improving the Index Compression

The techniques presented in the previous section were developed to compress inverted files or full inverted indexes. Special features of

Table I. Sample codes for integers.

Integer x	Unary	Elias-γ	Elias-δ
1	0	0	0
2	10	100	1000
3	110	101	1001
4	1110	11000	10100
5	11110	11001	10101
6	111110	11010	10110
7	1111110	11011	10111
8	11111110	1110000	11000000
9	111111110	1110001	11000001
10	1111111110	1110010	11000010

the block addressing indexes can be used to improve the compression without significant changes in the performance of the system.

In blocking addressing, many words can appear in more than half of the blocks. This phenomenon is not common in full inverted indexes or inverted files, but can occur frequently in block addressing indexes when large block sizes are used. In these cases, a simple idea to improve the index compression is to represent the list of non-occurrences of these more frequent words. That is, if a word occurs in more than half of the blocks then we store the block numbers where it does *not* occur. We will call these lists *complemented lists*.

An alternative form to compress those words would be to use run length compression on the gaps (which would be 1 at least half of the times). The economy of space is very similar because the length of each run of "ones" is precisely the value of the gap in the complemented list minus 1. For instance, if there are 100 blocks and the word appears in all but the 32nd and 61st, then its list of gaps is $[1, 1, ..., 1, 2, 1, 1, ..., 1, 2, 1..., 1]$. Run length compression on the list of gaps yields $\langle 1, 31\rangle\langle 2, 1\rangle\langle 1, 28\rangle\langle 2, 1\rangle\langle 1, 39\rangle$, in the format $\langle$ *number, repetitions* $\rangle$. On the other hand, the complemented list is $[32, 61]$, and the list of gaps is $[32, 29]$. Note that run length compression needs to store more information than that of the gaps in the complemented list.

A second advantage is that complemented lists can be operated upon efficiently without converting them into normal lists. We describe later how to perform Boolean operations among normal and complemented lists in time proportional to their normal or complemented representations. Depending on the operation, the result is left in normal or complemented form.

In inverted indexes it is common to not index the stopwords to save space. Since stopwords will most probably appear in all the blocks, we can index them at almost zero cost. Moreover, we need to keep them in the vocabulary for decompression purposes.

5.2. In-Memory Bucket Compression

In other compressed inverted schemes (Witten et al., 1999) the generation of the inverted list proceeds in a first stage and their compression in a second stage. This is not only because the compression is parametric in some cases, but also because of the way in which the lists are generated. In a first step, the text is traversed and the occurrences are generated in text order. Later, the occurrences are sorted by the word they represent. Therefore, only after this final sorting the lists are separated by words and the gaps can be generated in order to compress the lists.

As we are using a non-parameterized coding scheme, we do not need global information about the list in order to compress it. An additional decision that allows the lists to be generated in memory already in their compressed form is that we do not generate the occurrences in text order and later sort them, but we generate them already separated by word. To do this, we store a separate list of occurrences of each word (since we already know the vocabulary) and each new text occurrence is added at the end of the occurrence list of the corresponding word. Therefore, we can compute the gaps and store them in compressed form on the fly.

The technique of generating the occurrences in unsorted form first is sometimes preferred because of space reasons: storing a list of occurrences for each word may be a waste of space because either too many pointers have to be stored or too much empty space has to be preallocated. This is especially important because, by Zipf's Law, many words have very few occurrences. Storing separate lists, on the other hand, have the advantage of avoiding the final sort, which saves time. When combining this with compression, another advantage for separate lists appears: the lists can be generated in compressed form and therefore they take less space. This improved space usage translates also into better indexing time because more text can be indexed in memory without resorting to disk. Other indexing schemes that avoid sorting the occurrences are presented in (Hawking, 1997; Moffat, 1992; Harman et al., 1992).

We propose now an efficient approach to store the lists of occurrences of each word that tries to adapt to the typical word frequencies. The idea is to represent the list of occurrences of each word by using a linked list where each node is a bucket of occurrences. These buckets are composed by a pointer to the next bucket of the term and by a stream of bits that represents a portion of the compressed list of this term. The next bucket pointed has the same structure and continues the stream of bits.

An important decision in this scheme is the size of the buckets. They should be large enough to compensate the extra space used by the pointer to the next bucket, and should be small enough to reduce the extra memory lost with the empty spaces on the last bucket of each term. After some experiments, we have chosen to use 8 bytes for the first bucket of each term and 16 byte buckets for the remaining buckets of each term. The reason to use a smaller first bucket is that many terms can occur just once on the whole collection. So, using a smaller first bucket saves memory in these terms.

Figure 4 shows an example with the list of the occurrences of a term t that has appeared at the blocks $[1, 5, 10, 12, 14, 20, 30]$. Using the coding scheme shown in the last section, this list is converted in the list of *gaps* $[1, 4, 5, 2, 2, 6, 10]$. Using the Elias-γ coding scheme the list of *gaps* is converted into the stream of bits 01100011001100100110101110010. Using a first bucket size of 32 bits and the remaining buckets with 64 bits, the buckets for this term are as shown in Figure 4.

The empty space in the two buckets is the space to represent the pointer to the next bucket in the linked list. This pointer can be represented in $\lceil \log b \rceil$ bits, where b is the number of buckets that can fit in the memory buffer. In the example of Figure 4, these pointers were represented in 20 bits, allowing up to 2^{20} buckets in the main memory.

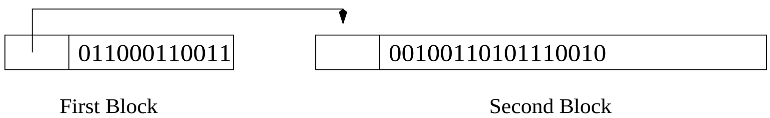


Figure 4. Linked list of buckets used with the in-memory compression scheme.

This in-memory bucket compression technique allows us to index large texts by making just one pass on the text to generate the index. It is general and can be applied in the construction of any kind of inverted index, such as in full inversion and inverted files. If the whole index cannot be placed in memory, we need to dump the partial list to disk and make a second pass to merge the dumps as described in (Moffat and Bell, 1995).

6. Putting All Together

We present in this section our combined design which includes text compression, block addressing and index compression into a single approach. The resulting structure is as follows (see Figure 5):

Vocabulary: first, we have the vocabulary of the whole collection, which is useful both as the symbol table of the Huffman coding in use and as the inverted index vocabulary. The canonical Huffman tree comprises a small extra structure which, added to the vocabulary, is all we need to search and decompress the text. Recall that there are also a few separators which are present in the Huffman tree but are not searchable.

Occurrences: for each vocabulary word we have a list of the blocks where the word appears. The list is in increasing block number and is compressed using the techniques of Section 5. Despite that separators are kept in the vocabulary for decompression purposes, we do not build their lists of occurrences. Another common choice in inverted indexes is to filter the words (e.g. map letters to lowercase, apply stemming, etc. (Baeza-Yates and Ribeiro-Neto, 1999)) which we cannot do here because we could not recover the original file. Instead, this filtration is done on the fly at search time.

Block structure: the blocks form a logical layer over the natural file structure of the collection, so that the files are not physically split or concatenated. This is implemented as a list of the files of the collection, so that the position of a file in that list is a sort of identifier. We also keep a list of the r blocks used. All the blocks have the same number of words and span a continuous range in the list of file identifiers, not necessarily matching the file boundaries. For each block we store the identifier of the last file that it spans in the list, and the offset of the last byte in that file that belongs to the block.

Text files: each original file in the collection is compressed as a separate file (although a single Huffman coding is used for the whole collection).

The space of this index is analyzed in (Baeza-Yates and Navarro, 2000), where it is shown that the vocabulary takes $O(n^\beta)$ space and the occurrences take $O(rb^\beta)$ space (since, by Heaps' Law, each new block has $O(b^\beta)$ different words, and one reference for each of them exists in the lists of occurrences). The lists of blocks and files are negligible in

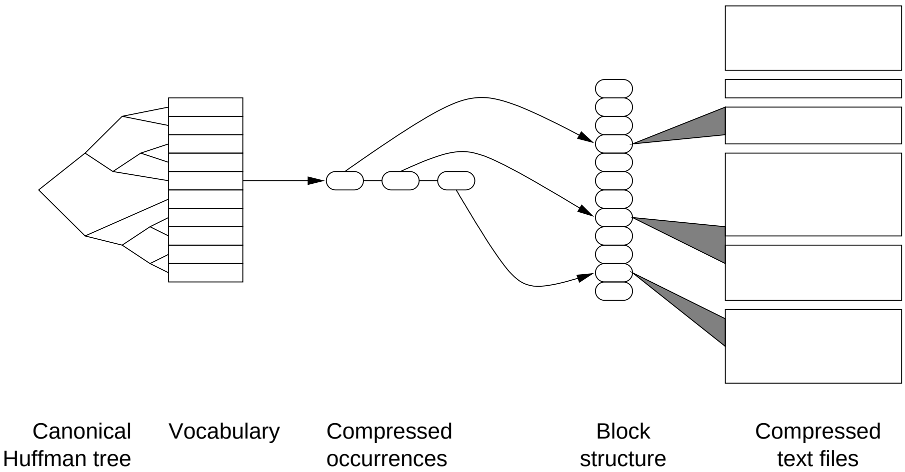


Figure 5. The structure of our index.

size. On the other hand, the occurrences are compressed now, which reduces their space by a factor independent of n (and therefore the space is still $O(rb^\beta)$). As shown in (Baeza-Yates and Navarro, 2000), the β value for small blocks is larger than its asymptotic value, but it converges very close to it for relatively small blocks.

6.1. Construction

The index can be efficiently built if we notice that many processes can be merged in a single pass over the text. The Huffman compression needs two passes over the text, and in the same two passes we are able to build the index. The index construction process has three stages.

6.1.1. *Stage 1*

The first stage corresponds to finding all the global information of interest. This is: determine the set of files to index and the number of blocks to use; compute the vocabulary of the whole collection and the frequencies of each word; and determine which lists will be represented by their complement. This requires a simple linear pass over the text, and the memory required is that of storing the vocabulary (the list of files can be output to disk as they are found). At the end of this pass, we have computed the list of files and the vocabulary of the text with its frequencies. At the end we also know the total number of words in the collection and therefore, we can define the number of blocks r.

Next, we need to collect two different frequency parameters. The first one is the number of times that each word occurs and the second one is the number of blocks in which each word occurs. The first one is needed by the Huffman algorithm, while the second one is used to determine whether the list of the word will be stored in simple or complemented form. These statistics are also useful for relevance ranking. While the number of times that a word occurs is easy to compute, the number of blocks requires a little care: we store for each word the last block where it appeared and the number of blocks where it already appeared. We know which is the current block because we increment it each time b new words are read. Hence, for each occurrence of a word we check whether it already appeared in the current block or not. In the second case, we increment the number of blocks where it appeared and update the last block where the word was seen.

Finally, before moving to the next stage we run the Huffman algorithm on the vocabulary and build the canonical Huffman tree. We replace the frequency information of each vocabulary word by the compressed code it has been assigned. The tree itself can immediately be swapped out to disk (although it is very small anyway). On the other hand, any data structure used to build the vocabulary, i.e. to efficiently find the words, should be kept in memory as it will be of help in the second stage. The vocabulary can be stored in memory by using a hash table or a trie in order to provide $O(1)$ average or worst-case access time, respectively.

6.1.2. *Stage 2*

The second stage does the heavier part. Each text file of the collection is compressed, in the order dictated by the list of files. Each word (or separator) of the text is searched in the vocabulary (this is why we need the data structures to search the words) and its Huffman code is output to the compressed file. When we finish with a file, the compressed version replaces the original one.

At the same time, we construct the lists of occurrences. Each time a word is found and we output its compressed code, we add an entry to its list of occurrences, which is represented as shown in Section 5.2. Of course the entry is not added if it has already appeared in that block, so we store for each word the last block it appeared in. Recall also that the inverse process is done for words whose occurrence list is to be stored in complemented form: if a word appears in a block and the previous block it appeared is neither the current nor the previous one, then we add to its list all the block interval between the current block and its last occurrence (all the last non occurrences of its word).

The current block number is incremented each time b new words are processed. At this time, we add a new entry to the list of blocks pointing to the current file being processed and store the number of bytes already written in the compressed version of the file. This list can be sent to disk as it is generated and will be used to map a block number to the physical position of the block in the collection.

At the end, the list of occurrences is sent to disk in its compressed form. Separately, we save the vocabulary on disk with pointers to the place where the list of occurrences of each word starts in the file of the occurrences.

The problem with the above scheme is that, despite that the index needs little space, we may not be able to store all the occurrence lists in memory at construction time. This is the only problematic structure, as the rest either is small enough or it can be buffered. The occurrences, on the other hand, cannot be sent to disk as they are generated because all the final entries of the first list should come before those of the second list. The solution chosen is that each time the memory is filled we store all the occurrences computed up to now on disk and free their space in memory, starting again with an empty list.

6.1.3. *Stage 3*

At the end of Stage 2, we will have a set of partial occurrence lists which have to be merged in the order given by the words they correspond to. All the lists of each word are concatenated in the order they were generated. So some auxiliary information has to be stored with the partial lists to help identify the word they belong to: a word identifier and the length of the list is enough.

6.1.4. *Analysis*

Collecting the vocabulary of the text can be done in linear time provided adequate data structures are used (e.g. a trie of the words to guarantee worst case linear time or a hash table for average linear time). Huffman construction can be done in linear expected time if the words follow some statistical rules widely accepted in text retrieval, as shown in (Moura et al., 1999). The other processes of the first stage are also of linear time and negligible in practice.

The second stage is also linear if we use the discussed data structures to find the text words in the vocabulary. The compressed codes output total less space than the text itself (so they take also linear time) and adding the block numbers to the end of the lists of occurrences is also constant time per text word.

What is not linear is the third stage that merges the lists. If we have $O(M)$ memory available for index construction, then $O(n/M)$ partial occurrence lists will be generated and will have to be merged. By using heapsort, the total merge takes $O(n \log(n/M))$ time. This third stage can be avoided by resorting to virtual memory, but writing partial lists to disk and merging them is much faster in practice. It is interesting to see that the merge phase will commonly not be needed because we use in-memory compression and block indexes tend to be small. For example, using a machine with 100 Mb of RAM and a 500 words block length (a small block size), we are able to index a collection size close to 1 Gb without needing Stage 3.

6.2. Searching

We describe now the search process using our index, which can be divided in three steps. We first explain the search of a single element and then show how to search phrases.

6.2.1. *Step 1*

The first step of the search is to find the query pattern in the vocabulary (be it a single word, a regular expression, allowing or not errors, etc.). The data structures used at indexing time can be kept to speed up this search, or we can resort to sequential or binary search to save space[2]. This is done exactly as explained in Section 3. At the end, we have a list of words that match the query, and we have built the binary masks for each of them (in case of phrase searching).

6.2.2. *Step 2*

This is where we take advantage of the block information, which cannot be done in simple sequential searching. The query pattern has been matched to a set of vocabulary words (just one if we search for a single word). We take the list of blocks where each of the words occur and merge all them into a single list, which is ordered by increasing block number. None of the blocks excluded from this final list can contain an occurrence of the query element.

Since the lists to merge are in compressed form we decompress them on the fly at the same time we merge them. For each new list element to read, we decode the bits of the compressed representation of the gap and add that gap to the last element of the list that has already been processed.

The other technique we used to reduce the size of the lists is the complementation of long lists. The operation on complemented lists can be done very fast, in time proportional to the complemented list. If two complemented lists ℓ_1^c and ℓ_2^c have to be merged, the complemented result is $(\ell_1 \cup \ell_2)^c = \ell_1^c \cap \ell_2^c$, i.e. we intersect their complements and have the complement of the result. Similarly, if they have to be intersected

[2] Note, however, that the words cannot be simultaneously sorted alphabetically and in the order required by the canonical Huffman tree, so at least an extra indirection array is required.

we apply $(\ell_1 \cap \ell_2)^c = \ell_1^c \cup \ell_2^c$. If ℓ_1 is complemented and ℓ_2 is not, then we proceed by set difference: $(\ell_1 \cup \ell_2)^c = \ell_1^c - \ell_2$ and $\ell_1 \cap \ell_2 = \ell_2 - \ell_1^c$.

6.2.3. *Step 3*

The final step is the sequential search on the blocks, to find the exact documents and positions where the query occurs. Only the blocks that are mentioned in the list of occurrences of the matched words need to be searched. The block structure is used to determine which portions of which files are to be sequentially traversed.

The search algorithm is exactly the same as for sequential searching without index. However, we have a new choice with respect to the multipattern Boyer-Moore search. In the sequential setup, all the compressed codes of the matching words are simultaneously searched, since there is no information of where each different word appears. It is clear that the search performance degrades as the number of patterns to search grows.

With the index we have more information. We know in which block each vocabulary word matched. Imagine that the query matched words w_1 and w_2. While w_1 appears in blocks b_1 and b_2, w_2 appears in blocks b_2 and b_3. There is no need to search w_2 in b_1 or w_1 in b_3. On the other hand, b_2 has to be searched for both words. We can therefore make a different (and hopefully faster) search in each text block. The price is that we need a different preprocessing for each block, which could be counterproductive if the blocks are very small. This idea is mentioned in (Baeza-Yates and Navarro, 2000), but not tested.

6.2.4. *Phrase search*

A query may not be just a pattern but a sequence of patterns, each one being a word, a regular expression, etc. The main idea to search phrases is to take the *intersection* of the occurrence lists of the involved blocks. This is because all the elements of the phrase must appear in the same block (we consider block boundaries shortly). We proceed as before with each pattern of the phrase: the list of occurrences of each pattern is obtained by making the union of all the list of the vocabulary words that match the pattern. Once we have the list for each pattern of the phrase, we intersect all the lists, and perform the sequential search only on the blocks where all the patterns appear at the same time. Unlike the case of simple elements, we may search blocks that have no occurrences of the complete query.

A natural question at this point is how can we avoid losing phrases that lie at block boundaries, since the intersection method will fail. This can be solved by letting consecutive blocks *overlap* in a few words. At indexing time we determine that we will allow searching phrases of at most ℓ words. Therefore, if a block ends at the i-th word of a document, the next one does not start at word $i+1$ but at $i+2-\ell$. This causes a very small overhead and solves elegantly the problem, since every phrase of ℓ words or less appears completely inside a block. The main problem is that we limit at indexing time the longest phrase that can be searched. For words longer than ℓ we can modify the list intersection process, so that two contiguous blocks are verified if the first words of a phrase appears in the first block and the last words in the second block. This, however, is much more expensive.

Another solution is to slightly relax the rule that the blocks are exactly b words long and move block boundaries a little to make them match with the end of a sentence (i.e. a separator containing a period). In this case no phrase can cross the block boundaries and there is no need to make blocks overlap or to limit beforehand the length ℓ of the phrases to search. On the other hand, parsing the text is a bit more complicated.

Even more sophisticated searching, such as proximity search (i.e. a sequence of ℓ words that must appear in correct order in an interval of ℓ' text words), can be accommodated in a very similar way. Only the sequential searching changes (see (Moura et al., 2000) for techniques to search diverse sophisticated patterns), and the block boundaries are made overlap in $\ell' - 1$ words.

6.2.5. *Analysis*

We now analyze the performance of the search process. The first step (vocabulary search) has already been analyzed in Section 3: a phrase of j elements takes $O(jn^{\beta})$ or $O(jkn^{\beta})$ time depending on the complexity of the search.

The second step is the merging and/or intersection of lists. First consider one-word queries, which is analyzed in (Baeza-Yates and Navarro, 2000) (recall Section 4). Since each word has an occurrence list of $O(n^{1-\beta})$ average length, the cost to merge the lists is $O(n^{1-\beta+\alpha} \log n)$. The cost to intersect the lists of a phrase of j such patterns is $O(jn^{1-\beta+\alpha} \log n)$, because since the lists are stored in compressed form we need to traverse all of them. Recall, however, that very long lists are compressed by representing their complement and these representations can be efficiently handled.

However, the cost of the first and second steps is negligible compared to that of the third step. Since we know already the search cost on a text of a given size, what remains to be determined is the percentage of text that has to be traversed when a block index is used. First we consider one-word patterns. Since a block of b words has $O(b^{\beta})$ different words, and $O(n^{\alpha})$ random words out of $O(n^{\beta})$ vocabulary words are selected by the search, the probability that the block gets a selected word and hence is searched is $O(b^\beta n^{\alpha-\beta})$. Since there are r blocks and the cost to traverse them is $O(b)$, we have that the total search cost is $O(brb^\beta n^{\alpha-\beta}) = O(n^{1-\beta+\alpha} b^\beta)$. When b tends to 1 the cost approaches that of full inverted files (Araújo et al., 1997).

Phrase searching is much better, however. As shown in (Araújo et al., 1997) using Zipf's Law, the shortest list among 2 or more random words has constant length. This means that on average we will search $O(1)$ blocks for phrase searching, which is $O(b)$ time. The cost to intersect the lists is similar to that of the union, because they are sequentially processed.

To summarize, the total search cost is $O(n^\beta + n^{1-\beta+\alpha} b^\beta)$ for single patterns and $O(n^\beta + n^{1-\beta+\alpha} + b)$ for phrases. We have considered k and j as constants to simplify the final complexity.

6.3. Updating

The final issue is how to update this index when documents are added, removed, or modified. Procedures to update a normal inverted index are well known (Brown et al., 1994), but there are extra complications with a block index.

Removing a document cannot be simply handled by removing all references to it in the occurrence lists, since we only point to blocks that contain the document or overlap with it (we could remove complete blocks if they are totally contained in the removed document). An alternative is to reindex the block, but it is expensive and the block is of different size now, which is not incorrect but may degrade the performance. Inserting new documents can be handled by adding new blocks as they are needed, although the document has to be compressed and indexed and its occurrence lists merged with those of the whole index. On the other hand, as shown in (Baeza-Yates and Navarro, 2000), the block size b should grow (sublinearly) as the text size grows.

The best choice to handle updates in this type of index is periodic reindexing (Manber and Wu, 1994). That is, the index is completely rebuilt at periodic intervals (say, daily or weekly). This choice imply some limitations on the update frequency and the maximal size of the database. In between, we handle the updates in a manner that makes them very light. This is paid with a slight degradation of the index performance between periodic rebuilds.

Deletions: the document is marked as deleted in the list of documents and physically deleted. Nothing else is altered in the index. When the block has to be sequentially traversed, that document is of course skipped. This makes it possible that the block is traversed for some words that are not anymore in it. On the other hand, removing files from the collection is very fast.

Insertions: there are two good choices. A first one is to compress the file, add it to a last incomplete block of the index or create a new block for it, and add the entry of this block to all the lists of the words that appear in the new document. This fully integrates the document in the collection but takes some time. Another choice is to add the identifier of the document to a special block which is not indexed and therefore is included in every sequential search. This is equivalent to having the new files not indexed and search them sequentially all the times until the next rebuild. This degrades slightly the performance but makes insertions very fast. The index can be forced to rebuild when the user determines it or when the extra block becomes too large.

Replacements: the best way to handle a replacement is as a deletion followed by an insertion.

Another complication comes from the fact that we are compressing the text. Even in the case of periodic rebuilds, we would like to avoid recompressing the whole text too frequently. Therefore, we try to handle incremental modifications to the Huffman codes. Not only do the changes alter the frequencies of the words (and hence alter the optimality of the code we are using) but also new words could appear that have no representation in the Huffman tree.

A first choice is to leave a placeholder in the tree. A fake word of frequency zero is created, and its node is used when new words appear. Each time the placeholder is used, a new one must be created. On the other hand, words that totally disappear can be discovered when their occurrence list becomes null, and their place in the tree can be left as a placeholder for new words to be added.

At the beginning one can expect that the words that appear and/or disappear have very low frequency, and therefore a technique of placeholders yields negligible degradations in compression. However, as the changes accumulate over the time, a mechanism must be devised to recover the optimality of Huffman codes. A first choice is periodic recompression of the database (the period can be quite large in practice, as shown in (Moffat et al., 1997)). A more challenging alternative is to perform small modifications in the tree to keep it optimal and to minimize the number of files that need to be recompressed. This is an open research issue that we are pursuing.

An alternative method studied in (Moffat et al., 1997) is the use of escape codes. New words without representation in the Huffman tree are represented explicitly in the text (preceding them with a special byte). This scheme also needs a method to avoid compression degradation as changes accumulate along time. They show in (Moffat et al., 1997) that this is also a good idea for all the words with very low frequency, since putting them in the tree does not improve the compression, and taking them out of the tree reduces a lot the vocabulary.

This, however, does not merge well with an inverted file scenario, since we should keep anyway the words in the vocabulary to avoid traversing the whole text for each word not found in the (incomplete) vocabulary. Even worse, those unfrequent words are the most interesting ones for IR and the most frequently searched. Finally, the vocabulary sizes should not be a problem at all in current computer servers.

7. Experimental Results

We present in this section some experimental results to evaluate the effectiveness of the combined approach. For the tests we have used literary texts from the TREC collection (Harman, 1995). We have chosen one gigabyte from the following texts: AP - Newswire (1989), DOE - Short abstracts from DOE publications, FR - Federal Register (1989), WSJ - Wall Street Journal (1987, 1988, 1989) and ZIFF - articles from *Computer Selected* disks (Ziff-Davis Publishing). All the collection is put in a single file. We considered a word as a contiguous string of characters in the set $\{A\ldots Z, a\ldots z, 0\ldots 9\}$ separated by other characters not in the set $\{A\ldots Z, a\ldots z, 0\ldots 9\}$. All the tests were run on a PC Pentium II 400 MHz with 256 megabytes of RAM running Linux and not performing other tasks. The disk uses an interface SCSI ULTRA DMA-33. We show elapsed times.

7.1. Construction Time and Space

We start by evaluating the time and space to build the index as a function of the block size, to determine the effectiveness of the in-memory compression technique. Figure 6 shows the time and main memory requirements to construct the index and compress the texts of the collection when varying the block size from 500 to 10000 words. The RAM requirements include all the structures used by our implementation: the space to store the vocabulary, the data used by the compression algorithm and the space used to keep the list of buckets for each term of the collection.

As can be seen, the index for 1 Gb can be built in around 11 minutes and uses 90 to 140 Mb of RAM (to build it with less RAM we would need to build partial indexes and merge them). The differences when the block size grows are due to the reduction in the number of entries of the index. Just compressing the text takes about 8 minutes, but just indexing would need about 10 minutes, as the main cost is the text parsing and this is shared by the compressor and the indexer.

We have also included the time to index the same collection using MG (the block size is controlled by adding artificial document terminators inside the unique file at appropriate intervals). As can be seen, MG is 3.5 to 4 times slower. This can be due to diverse facts, such as (1) MG uses a set of separate programs coordinated with a shell script, which slows down execution; (2) MG does not use in-memory list compression and hence it needed to merge partial indexes.

Glimpse is not included in this comparison because we cannot control the block size except by actually splitting the input in real files. This slows down the indexing time at a point that makes any comparison useless.

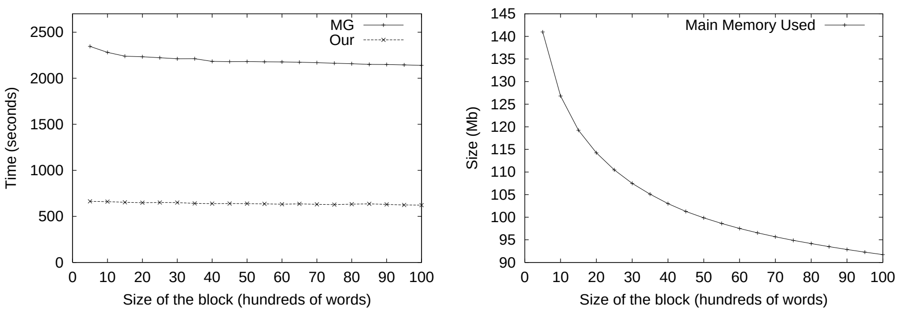


Figure 6. Time to build different indexes in seconds (left) and main memory usage in Mb for our index (right), for varying block size.

7.2. Compression Ratio and Space Overhead

We consider now the space overhead of the final index. Figure 7 (left) shows the number of bits per entry used under Elias-δ and Elias-γ. This decreases as the block size grows because the *gap* values tend to be smaller for larger blocks, which reduces the number of bits to represent the index entries by using Elias coding. The plot shows that there is little difference between both methods. We use Elias-γ from now on.

The right part of Figure 7 shows the size of the index as a function of the block size. The index size reduces quickly as the block size grows due to two main reasons. The first one is the reduction of bits per entry explained above. The second is that the number of index entries is reduced as the block size grows.

The right plot also shows the gain obtained by using complemented lists, which is modest.

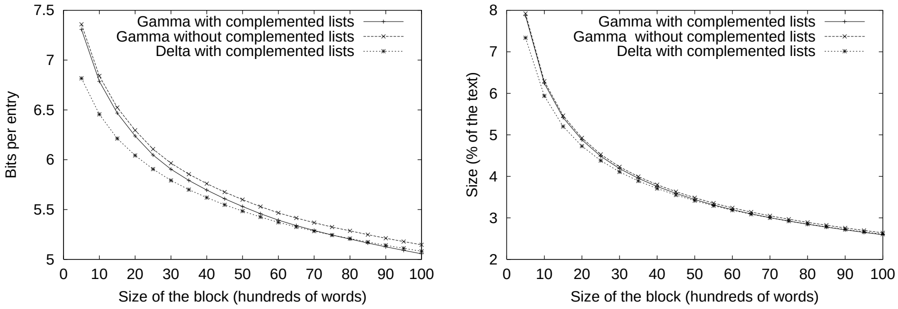


Figure 7. Bits per entry (left) and size of the compressed index as a percentage of the uncompressed text (right) when varying the block size.

7.3. Search Time

We consider the search time now. We measured it by randomly choosing patterns from the text. We have experimented patterns with 1, 2 and 3 words, averaging over 40 patterns of each pattern size. Figure 8 shows the time for exact and approximate searching with block sizes varying from 500 to 10000 words. Our sequential search algorithm used is the fast filter described in Section 3.2, where the false matches problem is solved by using synchronization points.

Search times do not include loading the vocabulary, so they correspond to a running server which has the vocabulary already loaded in memory and answers requests from users (warm state).

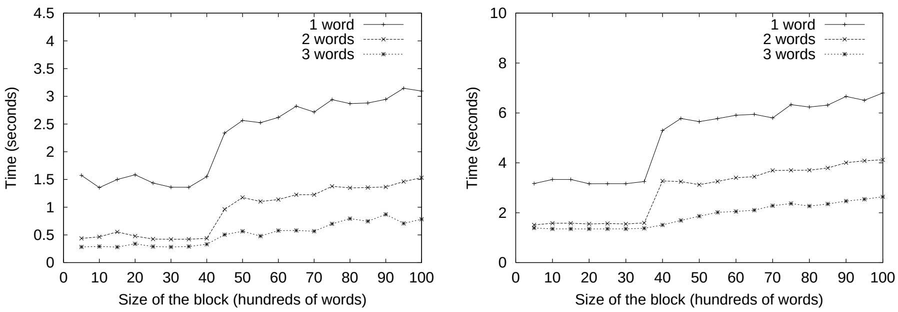


Figure 8. Time in seconds for exact searching (left) and allowing one error (right) for patterns with 1,2 and 3 words and varying the block sizes.

Figure 9 shows the amount of text traversed for the same experiments. As can be seen, a small percentage of the text is traversed even for a very small index. In particular, the search time and percentage of traversed text drops quickly when we search for phrases, since the number of blocks where all the involved words appear is much smaller.

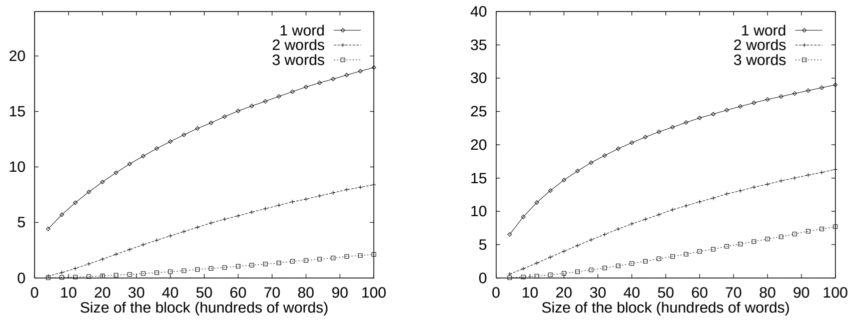


Figure 9. Amount of text traversed (in megabytes) for exact searching (left) and allowing one error (right) for patterns with 1,2 and 3 words and varying the block sizes.

To compare these figures against related work, we consider the "small" Glimpse index built over the 1 Gb text. In this case Glimpse produces an index that is 7% of the text size. We build our index to have the same space overhead, which gave us XXXX1683 blocks. That is, since our index takes less space, we can have smaller blocks for the same space overhead as Glimpse. This translates into better search time. For the construction of this index Glimpse took XXXX6.9 minutes and our index XXXX5.7 minutes (this includes the text compression). **[EDLENO]**

We also include in the comparison two sequential search algorithms, to show the gains due to the index: *Agrep* (Wu and Manber, 1992), a well known sequential searching software which is the base of Glimpse, and *Cgrep* (Moura et al., 2000), a software for sequential searching on Byte-Huffman compressed text (which is the sequential search algorithm our index uses). The results are shown in Table II. The timing methodology is as before, except that these times do include the time to load the vocabulary into memory (cold state). Notice that the search time of the indexes improve a lot on phrases, since only the blocks where all the words occur are searched.

7.4. Updating the Text

We finish by showing some experiments about the behavior of the compression ratios when the text is updated. From the techniques discussed in Section 6.3, we have implemented the one leaving a special escape

Table II. Comparison of search time (in seconds for 1 Gb).

Words	Agrep	Cgrep	Glimpse (7%)	Our index (7%)
Exact searching				
1	18.82	11.35	12.97	1.57
2	17.00	10.52	5.70	0.43
3	16.81	9.94	0.98	0.28
With 1 error				
1	71.01	12.37	40.36	3.17
2	72.01	11.73	14.94	1.51
3	67.05	11.45	6.82	1.39

symbol in the Huffman tree as an escape code, and code the new strings using a foxed-length code of 3 bytes following the escape symbol. A list of the newly inserted words is kept and the search algorithms can be altered to account for the escape codes.

Figure 10 (left) shows how deletions affect the compression quality. The different TREC texts under considerations were first compressed separately. Then, we removed text from each of the collections until they become of 10 Mb each. As can be seen, the compression ratios keep basically constant, which means that the frequency of the words in the total texts are approximated very well by those in a small fraction of the texts.

The right part of Figure 10 shows how the successive insertions affect the compression quality in the *wsj* file. We first compress, respectively, 10%, 40% and 70% of the file and then insert the rest using the already existing codes (and escape codes for the new words). The curve of 100% permits comparing against the optimal compression. For example, we obtain about 30% compression for the whole *wsj*, but if we build the Huffman codes for only the first 10% of the text and then insert the rest using those codes, the compression ratio raises to 30.7%. This is certainly a modest price.

Finally, we test a very pessimistic case, where the Huffman codes are built for a collection and then this text is completely deleted and a whole new collection is inserted. Figure 11 shows the compression ratios obtained. As can be seen, there are increments of abour 30% in the compression ratios.

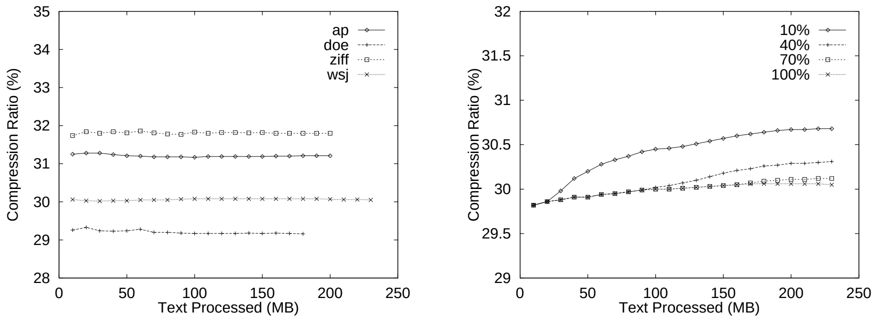


Figure 10. Compression ratios obtained after deleting portions of the files (left) or after inserting more text in *wsj* (right).

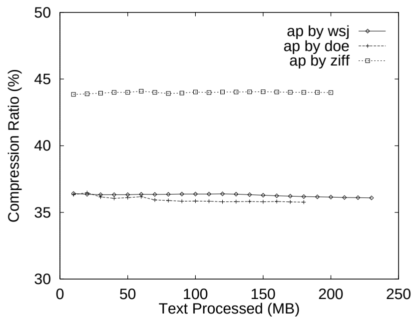


Figure 11. Compression ratios obtained after replacing a whole collection by another.

8. Conclusions

In this paper we have shown how to combine three different ideas that allow reducing the size of inverted indexes and the texts associated to them. These are block addressing, inverted index compression, and compressed text which permits direct search. In particular we have proposed new index compression techniques that are specific of block addressing indexes. The integration of these ideas still needs fine tuning as there are many parameters involved, but our experimental results suggest that block size should be around 4,000 words for 1 Gb. At this point we have a reasonable trade-off between index space and search time.

Using a block size of 4,000 words, the index just needs [**EDLENO**] XXX8Mb (XXXX4% of the text size) and could even be cached in memory. In addition, during construction time, only XXX20Mb are needed (XXX10% of the text size). On the other hand, the search time is less than 1.6 seconds for one word, and less than 0.5 seconds for more words, which is quite reasonable. The percentage of text traversed with this space overhead is around XX%-YY%.[**EDLENO WILL SEND**

AFTER EXPERIMENTS] The overall result is that the index and the text can be compressed into less than 40% of the original text size alone, achieving searching times of a few seconds for 1 Gb. We have also shown that our index is up to 8 times faster than Glimpse.

A number of issues are still open, such as

Boolean queries: how to efficiently do Boolean operations in our inverted indexes? Document addressing has clear advantages here, as the operation can be translated into set operations on the lists of occurrences. Our smaller index can still be used to operated lists of blocks as a pre-filter to reduce the text that has to be scanned.

Structured text: can we add support for structural queries, at least to limit the occurrences to occur inside a given text field? As shown in (Baeza-Yates and Navarro, 1996; Baeza-Yates et al., 1998), inverted files can significantly help in supporting structured text queries by indexing (real or virtual) structural tags as distinct words and translating structural queries into operations regarding words or tags followed by other words or tags. A block addressing index can, in the same way, help filter out blocks that cannot contain such an occurrence.

Flexible pattern matching: how can we use the block addressing index to filter out text for more sophisticated pattern matching, such as approximate searching at the word level, described in (Moura et al., 2000)?

Efficient updating: how to reflect in the index the updates that the text collection undergoes? Although periodic rebuilding works, this cannot be done for very large collections (e.g., several gigabytes). For example, storage management techniques that allow for expansion and contraction of the occurrence lists are of interest.

A different integration of these ideas appears when considering the problem of searching the Web. In this case, block addressing can be used to distribute the search space in size and computation load. Hence, a central search engine handles a smaller index than current indexes (e.g. AltaVista), allowing more scalability than pure centralized architectures. This server distributes the query to a small number of local servers that have a local index. Therefore, in this case the block size is large, is not fixed, and it is not unique. Local indexes may or may not use again block addressing, and may use or not text compression. If they do, we have one vocabulary per block. This integration has the advantage of being more flexible, that some decisions are local (for example, the use of compression may depend on other factors), and that the architecture of the system is scalable. The main disadvantages are that this needs the cooperation between different sites (this is not a problem if it is done by a single company) and that network traffic increases. This idea is pursued in (Baeza-Yates, 2000) and is related to Harvest (Bowman et al., 1994). **[RICARDO: ref, mejorar o sacar]**

References

Araújo, M. D., G. Navarro, and N. Ziviani: 1997, 'Large text searching allowing errors'. In: R. Baeza-Yates (ed.): *Proc. of the 4th South American Workshop on String Processing*, Vol. 8. pp. 2–20.

Baeza-Yates, R.: 2000, 'Another Distributed Searching Architecture for the Web'. In preparation.

Baeza-Yates, R. and G. Gonnet: 1992, 'A new approach to text searching'. *Communications of the ACM* **35**(10), 74–82.

Baeza-Yates, R. and G. Navarro: 1996, '*Integrating contents and structure in text retrieval*'. *ACM Special Interest Group in Management of Data (SIGMOD) Record* **25**(1), 67–79.

Baeza-Yates, R. and G. Navarro: 2000, 'Block-Addressing Indices for Approximate Text Retrieval'. *Journal of the American Society for Information Science (JASIS)* **51**(1), 69–82.

Baeza-Yates, R., G. Navarro, J. Vegas, and P. de la Fuente: 1998, 'A Model and a Visual Query Language for Structured Text'. In: *Proc. of the 5th South American Symposium on String Processing and Information Retrieval (SPIRE'98)*. pp. 7–13.

Baeza-Yates, R. and M. Régnier: 1990, 'Fast Algorithms for Two Dimensional and Multiple Pattern Matching'. In: *Proc. of the 2nd Scandinavian Workshop on Algorithm Theory (SWAT'90)*. pp. 332–347.

Baeza-Yates, R. and B. Ribeiro-Neto: 1999, *Modern Information Retrieval*. Addison-Wesley.

Bell, T. C., J. G. Cleary, and I. H. Witten: 1990, *Text compression*. Prentice Hall.

Bentley, J., D. Sleator, R. Tarjan, and V. Wei: 1986, 'A locally adaptive data compression scheme'. *Communications of the ACM* **29**, 320–330.

Bowman, C., P. Danzig, D. Hardy, U. Manber, and M. Schwartz: 1994, 'The Harvest Information Discovery and Access System'. In: *Proc. of the 2nd International World Wide Web Conference*. pp. 763–771.

Brown, E. W., J. P. Callan, and W. B. Croft: 1994, 'Fast Incremental Indexing for Full-Text Information Retrieval'. In: *Proc. of the 20th Very Large Data Base Conference (VLDB'94)*. Santiago, Chile, pp. 192–202.

Elias, P.: 1975, 'Universal codeword sets and representations of the integers'. *IEEE Transactions on Information Theory* **IT-21**, 194–203.

Golomb, S. W.: 1966, 'Run-length encodings'. *IEEE Transactions on Information Theory* **IT-12**(3), 399–401.

Harman, D., E. Fox, R. Baeza-Yates, and W. Lee: 1992, *Inverted files*, pp. 28–43. Prentice-Hall.

Harman, D. K.: 1995, 'Overview of the Third Text REtrieval Conference'. In: *Proc. of the 3rd Text REtrieval Conference (TREC-3)*. Gaithersburg, Maryland, pp. 1–19.

Hawking, D.: 1997, 'Scalable Text Retrieval for Large Digital Libraries'. In: C. Peters and C. Thanos (eds.): *Proc. of the 1st European Conference on Digital Libraries.* Pisa, Italy, pp. 127–146.

Heaps, H.: 1978, *Information Retrieval - Computational and Theoretical Aspects.* NY: Academic Press.

Huffman, D. A.: 1952, 'A method for the construction of minimum-redundancy codes'. In: *Proc. of the Institute of Electrical and Radio Engineers*, Vol. 40. pp. 1090–1101.

Linoff, G. and C. Stanfill: 1993, 'Compression of Indexes with Full Positional Information in Very Large Text Databases'. In: *Proc. of the International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'93).* pp. 88–95.

Manber, U. and S. Wu: Winter 1994, 'GLIMPSE: A tool to search through entire file systems'. In: *Proc. of the USENIX Technical Conference.* pp. 23–32.

Moffat, A.: 1989, 'Word-based text compression'. *Software Practice and Experience* **19**(2), 185–198.

Moffat, A.: 1992, 'Economical Inversion of Large Text Files'. *Computing Systems (USENIX Assoc. Journal)* **5**(2), 125–139.

Moffat, A. and T. Bell: 1995, 'In-situ generation of compressed inverted files'. *Journal of the American Society for Information Science* **46**(7), 537–550.

Moffat, A., J. Zobel, and N. Sharman: 1997, 'Text Compression for Dynamic Document Databases'. *IEEE Transactions on Knowledge and Data Engineering* **1**(2).

Moura, E., G. Navarro, and N. Ziviani: 1999, 'Linear time sorting of skewed distributions'. In: *Proc. of the 6th Symposium on String Processing and Information Retrieval (SPIRE'99).* pp. 135–140.

Moura, E., G. Navarro, N. Ziviani, and R. Baeza-Yates: 2000, 'Fast and Flexible Word Searching on Compressed Text'. *ACM Transactions on Information Systems.* To appear. Previous versions in *21st International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'98)* and *5th Symposium on String Processing and Information Retrieval (SPIRE'98).*

Turpin, A. and A. Moffat: 1997, 'Fast file search using text compression'. In: *Proc. of the 20th Australian Computer Science Conference.* pp. 1–8.

Williams, H., J. Zobel, and P. Anderson: 1999, 'What's Next? Efficient Structures for Phrase Querying'. In: J. Roddick (ed.): *Proc. of the 10th Australasian Database Conference, Auckland, NZ.* Singapore, pp. 141–152.

Witten, I., A. Moffat, and T. Bell: 1999, *Managing Gigabytes.* New York: Morgan Kaufmann Publishers, second edition.

Wu, S. and U. Manber: 1992, 'Fast text searching allowing errors'. *Communications of the ACM* **35**(10), 83–91.

Zipf, G.: 1949, *Human Behaviour and the Principle of Least Effort.* Addison-Wesley.