Adding Compression to Next-Generation Text Retrieval Systems

Nivio Ziviani
Department of Computer Science
Univ. Federal de Minas Gerais, Brazil

Edleno Silva de Moura
Department of Computer Science
Univ. Federal do Amazonas, Brazil

March 14, 2003

Abstract

In this chapter we present a combination of several data compression features to provide economical storage, faster indexing, and accelerated searches. We discuss recent methods for compressing the text and the index of text retrieval systems. By compressing both the complete text and the index, the total amount of space is less than half the size of the original text alone. Most surprisingly, the time required to build the index and also to answer a query is much less than if the index and text had not been compressed. This is a rare case where there is no space-time trade-off. Moreover, the text can be kept compressed all the time, allowing updates when changes occur in the compressed text.

Keywords: Text retrieval systems, text and index compression.

Contents

1 Introduction

The widespread use of digital libraries, office automation systems, document databases, and lately the World-Wide Web has led to an explosion of textual information available online [6, 45]. In [22], the Web alone was shown to have approximately 800 million static pages, containing a total of approximately 6 terabytes of plain text. A *terabyte* is a bit more than one million of million bytes, enough to store the text of a million books. In June 2002 the search engine Google [11] claims to have more than 2.5 billion static pages in its database. Text retrieval is the kernel of most Information Retrieval (IR) systems, and one of the biggest challenges to IR systems is to provide fast access to such a mass of text.

In this chapter we discuss recent techniques that permit fast direct searching on the compressed text and how the new techniques can improve the overall efficiency of IR systems. Traditionally, compression techniques have not been used in IR systems because the compressed texts did not allow fast access. However, recent text compression methods [33, 50] have enabled the possibility of searching directly the compressed text *faster* than in the original text, and have also improved the amount of compression obtained. Direct access to any point in the compressed text has also become possible, and the IR system might access a given word in a compressed text without the need to decode the entire text from the beginning. The new features lead to a win-win situation that is raising renewed interest in text compression for the implementation of IR systems.

Text compression is about finding ways to represent actual text in less space. This is accomplished by substituting the symbols in the text by equivalent ones that are represented using a smaller number of bits or bytes. For large text collections, text compression appears as an attractive option for reducing costs. The gain obtained from compressing text is that it requires less storage space, it takes less time to be read from disk or transmitted over a communication link, and it takes less time to search. The price paid is the computing cost necessary to code and decode the text. This drawback, however, is becoming less and less significant as technology progresses. In the last 20 years the time to access disks kept approximately constant while processing speed increased approximately 2,000 times [37]. As time passes, investing more and more computing power in compression in exchange for less disk or network transfer times becomes a profitable option.

The savings of space obtained by a compression method is measured by the *compression ratio*, defined as the size of the compressed file as a percentage of the uncompressed file. Besides the economy of space, there are other important aspects to be considered, such as compression and decompression speed. In some situations, decompression speed is more important than compression speed. For instance, this is the case with textual databases and documentation systems in which it is common to compress the text once and to read it many times from disk. Another important characteristic of a compression method is the possibility of performing pattern matching in a compressed text without decompressing it. In this case, sequential searching can be speeded up by compressing the search key rather than decoding the compressed text being searched. As a consequence, it is possible to search faster on compressed text because fewer bytes have to be scanned.

Efficient text retrieval on large text collections requires specialized indexing techniques. An *index* is a data structure built on the text collection intended to speed up queries. A simple and popular indexing structure for text collections is the inverted file [6, 45]. Inverted files are especially adequate when the pattern to be searched for is formed by simple words. This is a common type of query, for instance, when searching the Web for pages that include the words "text" and "compression". An *inverted file* is typically composed of a vector containing all the distinct words in the text collection (which is called the *vocabulary*) and a list of all document numbers in which each distinct word occurs (sometimes its frequency in each document is also stored). The largest part of the index is the lists of document numbers, for which specific compression methods have been proposed that provide very good compression ratios. In this case, both index construction time and query processing time can be significantly improved by using index compression schemes.

In the next section we review traditional and recent methods for compressing text. Following, we discuss how recent techniques permit fast direct searching on the compressed text. Next, we present compression methods for compressing the index. Later on, we show how the new techniques can improve the overall efficiency of IR systems. Next, we show how the text can be kept compressed all the time, allowing updates when changes occur in the compressed text. Finally, we discuss the trends and research issues and present a bibliographic discussion.

2 Text Compression Methods for IR Systems

One well-known coding strategy is Huffman coding [20]. The idea of Huffman coding is to compress the text by assigning shorter codes to symbols with higher frequencies. This can be achieved by assigning a unique variable-length bit encoding to each different symbol of the text. The 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. Words are the atoms on which most IR systems are built. Taking words as symbols means that the table of symbols in the compression coder is exactly the vocabulary of the text, allowing a natural integration between an inverted file and a word-based Huffman compression method. Word-based Huffman methods allow random access to words within the compressed text, which is a critical issue for an IR system. Moreover, character-based Huffman methods are typically able to compress English texts to approximately 60% while word-based Huffman methods are able to reduce them to just over 25%, because the distribution of words is much more biased than the distribution of characters.

Another important family of compression methods, called Ziv-Lempel, substitutes a sequence of symbols by a pointer to a previous occurrence of that sequence. Compression is obtained because the pointers need less space than the phrase they replace. The Ziv-Lempel family can be classified in two sub-groups. The first one is derived from a method published by Ziv and Lempel in 1977 [48], which is known as LZ77. The compression methods derived from the LZ77 obtain good compression ratios for a large variety of data sources and are extremely fast on decoding. Their main drawback is that the compression is slower than the decompression. For this reason, algorithms derived from the LZ77 are the preferred choice for situations where the data is compressed once and decompressed more frequently. Examples of applications which are based on the LZ77 method are the the Gzip, which is popular in UNIX operating systems, and the Winzip, which is popular in Windows operating systems.

The other subgroup of the Ziv-Lempel family is derived from a method presented by Ziv and Lempel in 1978, which is known as LZ78. In most practical applications, the compression methods based on the LZ78 obtain compression ratios slight worse than the ones obtained by methods derived form the LZ77. However, the compression and decompression processes are performed faster in LZ78 than in the LZ77. This characteristic makes the LZ78 based methods the preferred choice in applications where time is a critical factor in performance, such as data transmission operations. Two examples of applications based on LZ78 are the utility Compress, available in UNIX systems, and the compression methods used in modem data transmission, which in general use a variation of LZ78 proposed by Welch in 1984 [44], which is known as LZW.

Ziv-Lempel methods are popular for their speed, economy of memory and good performance, but they present important disadvantages in an IR environment. First, they require decoding to start at the beginning of a compressed file, which makes random access expensive. Second, they are difficult to search without decompressing. A possible advantage is that they do not need to store a table of symbols as a Huffman-based method does, but this has little importance in IR scenarios because the vocabulary of the text is needed anyway for indexing and querying purposes. Arithmetic coding [39, 46], another well-known compression method, presents similar problems.

Arithmetic coding [39, 46], another well-known compression method, presents similar problems. Since we here are interested in applications of compression methods that are important for IR systems, we will not present further discussions about Ziv-Lempel methods or arithmetic coding. A reader interested in more details about Ziv-Lempel methods can find it in [48, 49, 7]. A reader more interested in Arithmetic coding can find it in [45].

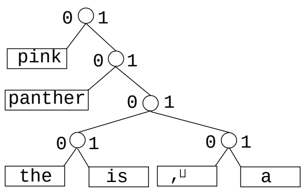
2.1 Word-Based Huffman Compression

For natural language texts used in an IR context, the most effective compression technique is word-based Huffman coding. 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 then it performs the actual compression in a second pass.

The text is not only composed of words but also of the strings which appears between the words, which we will name here as separators. An efficient way to deal with words and separators is to use a model called *spaceless words* [33]. This model takes in account that most of all the separators in a natural language text are composed by a single blank space. The spaceless model takes advantage of this fact by representing the spaces implicitly in the compressed text. In this model 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 {`"the"`, `"pink"`, `"panther"`, `"is"`, `",␣"`, "a" }, whose frequencies are 1, 4, 2, 1, 1, 1, respectively.

The example also shows how the codes for the symbols are organized in a *Huffman tree*. The most frequent word (in this case, `"pink"`) receives the shortest code (in this case, `"0"`). The Huffman method gives the coding tree that minimizes the length of the compressed file, but many coding trees would have achieved the same compression. For instance, exchanging the left and right children of a node yields an alternative coding tree with the same compression ratio.

When dealing with large vocabularies, the preferred choice for most applications is the *canonical tree*. A Huffman tree is canonical when the height of the left subtree of any node is never smaller than that of the right subtree, and all leaves are in increasing order of probabilities from left to right. The tree presented Figure 1 is canonical. The deepest leaf



Original text: `the pink panther is pink, a pink pink panther`

Compressed text: `1100 0 10 1101 0 1110 1111 0 0 10`

Figure 1: Compression using Huffman coding for spaceless words.

at the leftmost position of the Huffman canonical tree, corresponding to one element with smallest probability, will contain only zeros, and the following codes will be in increasing order inside each level. At each change of level we shift left one bit in the counting. Table 1 shows the canonical codes for the example of Figure 1.

Symbol	Probabilities	Canonical codes
`the`	1/10	`1100`
`is`	1/10	`1101`
`,␣`	1/10	`1110`
`a`	1/10	`1111`
`panther`	2/10	`10`
`pink`	4/10	`0`

Table 1: Canonical code.

A canonical code can be represented by an ordered sequence S of pairs (x_i, y_i), $1 \leq i \leq \ell$, where x_i represents the number of symbols at level i, y_i represents the numerical value of the first code at level i, and ℓ is the height of the tree. For the example in Figure 1, the ordered sequence is $S = \langle(1,0),(1,2),(0,\infty),(4,12)\rangle$. For instance, the fourth pair $(4,12)$ in S corresponds to the fourth level and indicates that there are four nodes at this level and that to the node most to the left is assigned a code, at this level, with value 12. Since this is the fourth level, a value 12 corresponds to the codeword 1100.

One of the properties of canonical codes is that the set of codes having the same length are the binary representations of consecutive integers. Interpreted as integers, the 4-bit codes in Table 1 are 12, 13, 14, and 15, the 2-bit code is 2 and the 1-bit code is 0. Another interesting property is that, interpreting the code as integers, if c is the code for the first element of level l and d is the code for the last element of level m, where m is the biggest non empty level smaller than l, then $c = 2^{(l-m)} \times (d+1)$. In the example, the last element of level one has code 0, so the first element of level two has code $2^{(2-1)} \times (0+1) = 2$. As level three has no code and the last element of level two has value 2, so the first element of level four has value $2^{(4-2)} \times (2+1) = 12$.

In the example, if the first character read from the input stream is 0, a codeword has been identified and the corresponding symbol can be output. If this value is 1, a second bit is appended and the two bits are again interpreted as an integer and used to index the table and identify the corresponding symbol. Once we read '`11`' we know that the code has four bits and therefore we can read two more bits and use them as an index into the table. This fact can be exploited to enable efficient encoding and decoding with small overhead. Moreover, much less memory is required, which is especially important for large vocabularies.

The compact representation of canonical trees and its sorting properties allow more efficiency at decoding time with less memory requirement. A careful discussion about how to implement efficient coding and decoding programs using canonical trees is described in [18]. An efficient algorithm for building the Huffman tree from the symbol frequencies is described in [45], and can be done in linear time after sorting the symbol frequencies.

Decompression in Huffman trees 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 `"1100"` in the compressed file leads to the decompressed symbol `"the"`.

2.2 Byte-Oriented Huffman Coding

The original method proposed by Huffman is mostly used as a binary code. In [33], the code assignment is modified such that a sequence of whole bytes is associated with each word in the text. As a result, the maximum degree of each node is now 256. This version is called *plain Huffman code* in this chapter. An alternative use of byte coding is what we call *tagged Huffman code*, where only 7 of the 8 bits of each byte are used for the code (and hence the tree has degree 128). The eighth bit is used to signal the first byte of each codeword (which, as seen later, aids the search). For example, a possible plain code for the word `"blue"` could be the 3-byte code “`47 81 8`”, and a possible tagged code for the word `"blue"` could be “`175 81 8`” (where the first byte $175 = 47 + 128$). Experimental results 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 are faster with a byte-oriented Huffman code than with a binary Huffman code, because bit shifts and masking operations are not necessary.

Table 2 shows the compression ratios and the compression and decompression times achieved for binary Huffman, plain Huffman, tagged Huffman, *gnu Gzip* and Unix *Compress* for the file WSJ containing the Wall Street Journal (1987, 1988, 1989), part of the TREC 3 collection [16]. The WSJ file has 250 megabytes, almost 43 million words and nearly 200,000 different words (vocabulary). As it can be seen, the compression ratio degrades only slightly by using bytes instead of bits. The increase in the compression ratio of the tagged Huffman code is approximately 3 points over that of the plain Huffman code, which comes from the extra space allocated for the tag bit in each byte. The compression time is 2-3 times faster than Gzip and only 17% slower than Compress (which achieves much worse compression ratios). Considering decompression, there is a significant improvement when using bytes instead of bits. Using bytes, both tagged and plain Huffman are more than 20% faster than Gzip and three times faster than Compress.

One important consequence of using byte Huffman coding is the possibility of performing fast direct searching on compressed text. A search algorithm is explained in the next section. The exact search can be done on the compressed text directly, using any known sequential pattern matching algorithm. The search algorithm presented allows a large number of variations of exact and approximate searching, such as phrases, ranges, complements, wild cards and arbitrary regular expressions. This technique is not only useful to speed up sequential

Method	Compression Ratio (%)	Compression Time (min)	Decompression Time (min)
Binary Huffman	27.13	8.77	3.08
Plain Huffman	30.60	8.67	1.95
Tagged Huffman	33.70	8.90	2.02
Gzip	37.53	25.43	2.68
Compress	42.94	7.60	6.78

Table 2: Comparison of compression techniques on the WSJ text collection.

search. As we see later, it can also be used to improve indexed schemes that combine inverted files and sequential search [35, 50, 45].

3 Online Search of Compressed Text

One of the most attractive properties of the Huffman method oriented to bytes rather than bits is that it can be searched exactly like any uncompressed text. When a query is submitted, the text is in compressed form and the pattern is in uncompressed form. The key idea is to compress the pattern instead of uncompressing the text. We call this technique *direct searching*.

The algorithm to find the occurrences of a single word starts by searching it in the vocabulary, where binary searching is a simple and inexpensive choice. Once the word has been found, its compressed code is obtained. Then, this compressed code is searched in the text using *any* classical string matching algorithm with no modifications. This is possible because the Huffman code uses bytes instead of bits, otherwise the method would be complicated.

A possible problem with this approach is that the compressed code for a word may appear in the compressed text even if the word does not appear in the original text. This may happen in plain Huffman codes because the concatenation of the codes of other words may contain the code sought, but it is impossible in tagged Huffman code [33]. The extra bit in the tagged Huffman code is employed to mark the starting point of each word in the compressed text, avoiding false matches during the searching process.

A common requirement of today's IR systems is flexibility in the search patterns. We call those "complex" patterns, which range from disregarding upper or lower case to searching for regular expressions and/or "approximate" searching. Approximate string searching, also called "searching allowing errors", permits at most k extra, missing or replaced characters between the pattern and its occurrence.

If the pattern is a complex word, we perform a *sequential* search in the vocabulary and collect the compressed codes of *all* the words that match the pattern. A multi-pattern search for all the codes is then conducted on the text [33]. Sequential vocabulary searching is not expensive for natural language texts because the vocabulary is small compared with the whole text (0.5% is typical for large texts). On the other hand, this sequential searching permits extra flexibility such as allowing errors.

The final setup allows a large number of variants, which forms a language originally defined for *Agrep* [47].

- 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).

Flexible Pattern Matching

Direct searching is very efficient but difficult to extend to handle much more complex queries, formed by phrases of complex patterns that are to be searched allowing errors and/or are defined by regular expressions. A more general approach, which works also on plain Huffman codes, uses the Huffman tree to guide the search process. To understand how this approach works we will first study the search algorithm for simple words and then show progressively how the query can be extended to phrases formed by complex patterns.

The searching algorithm for a single word starts again in the vocabulary using binary search. Once the word has been found the corresponding leaf in the Huffman tree is marked. Next, the compressed text is scanned byte by byte, and at the same time the Huffman tree is traversed downwards, as if one were decompressing the text, but without generating it. Each time a leaf of the Huffman tree is reached, one knows that a complete word has been read. If the leaf has a mark then an occurrence is reported. Be the leaf marked or not, one returns to the root of the Huffman tree and resumes the text scanning. Figure 2 illustrates the algorithm for the pattern "`blue`", encoded as the 3-byte `47 81 8`. Each time this sequence of bytes is found in the compressed text the corresponding leaf in the Huffman tree is reached, reporting an occurrence. Complex patterns are, as before, handled by a sequential search in the vocabulary. This time we mark all the leaves corresponding to matching words.

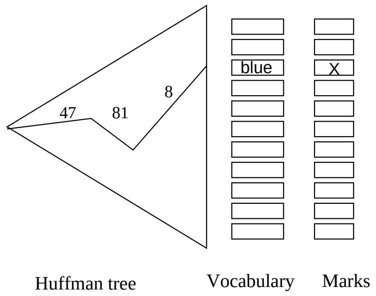


Figure 2: General searching scheme for the word "`blue`".

This simple scheme can be nicely extended to handle complex phrase queries. Phrase queries are a sequence of patterns, each of which can be from a simple word to a complex regular expression allowing errors. If a phrase has ℓ elements, we set up a mask of ℓ bits for each vocabulary word (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. For this sake, 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. Figure 3 illustrates the masks for the pattern "`bl* blue is`" allowing one error per word, where "`bl*`" means any word starting with "`bl`". The words "`blue`" and "`blues`" in the vocabulary match the pattern in the first and the second position. Therefore, their mask is "`110`" for the given pattern. The mask for the words "`is`" and "`in`" is "`001`", since they both match only the third position of the pattern. The mask for the word "`gray`" is "`000`", since it does not match any position of the pattern. A mask is assigned in a similar way for each work in the vocabulary.

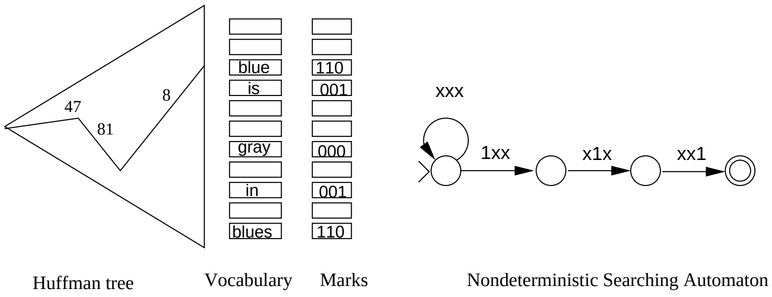


Figure 3: General searching scheme for the phrase "`bl* blue is`" allowing 1 error. The `x` in the automaton stands for 0 or 1.

After the preprocessing phase is performed the text is scanned as before. The state of the search is controlled by a nondeterministic automaton of $\ell + 1$ states, as shown in Figure 3 for the pattern "`bl* blue is`". The automaton allows to move from state i to state

$i+1$ whenever the i-th pattern 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. The bytes of the compressed text are read and the Huffman tree is traversed as before. Each time a leaf of the tree is reached its bit mask is sent 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.

This automaton can be implemented efficiently using the Shift-Or algorithm [3]. This algorithm is able to simulate an automaton of up to $w+1$ states (where w is the length in bits of the computer word), performing just two operations per entry. This means that it can search phrases of up to 32 or 64 words, depending on the machine. This is more than enough for common phrase searches. Longer phrases would need more machine words for the simulation, but the technique is the same.

The Shift-Or algorithm maps each state of the automaton (except the first one) to a bit of the computer word. For each new word reached in the Huffman three, each active state can activate the next one, which is simulated using a *shift* in the bit mask. Only those states that match the current text word 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).

It is simple to disregard separators in the search, so that a phrase query is found even if there are two spaces instead of one. Stop words (articles, prepositions, etc.) can also be disregarded. The procedure is just to ignore the corresponding leaves of the Huffman tree when the search arrives to them. This ability is common on inverted files but is very rare in online search systems.

Table 3 presents exact ($k=0$) and approximate ($k=1,2,3$) searching times for the WSJ file using *Agrep* [47], the direct search on tagged Huffman and the automaton search using plain Huffman. It can be seen from this table that both direct and automaton search algorithms are almost insensitive to the number of errors allowed in the pattern while Agrep is not. It also shows that both compressed search algorithms are faster than Agrep, up to 50% faster for exact searching and nearly 8 times faster for approximate searching. Notice that automaton searching permits complex phrase searching at exactly the same cost. Moreover, the automaton technique permits even more sophisticated searching, which is described below. However, automaton searching is always slower than direct searching, and should be used for complex queries as just described above.

Algorithm	$k=0$	$k=1$	$k=2$	$k=3$
Agrep	23.8 ± 0.38	117.9 ± 0.14	146.1 ± 0.13	174.6 ± 0.16
Direct Search	14.1 ± 0.18	15.0 ± 0.33	17.0 ± 0.71	22.7 ± 2.23
Automaton Search	22.1 ± 0.09	23.1 ± 0.14	24.7 ± 0.21	25.0 ± 0.49

Table 3: Searching times (in seconds) for the WSJ text file, with 99% confidence.

Enhanced Searching

The Shift-Or algorithm can do much more than just searching for a simple sequence of elements. For instance, it has been enhanced to search for regular expressions, to allow errors in the matches and other flexible patterns [47, 4]. This powerful type of search is the basis of the software Agrep [47].

A new handful of choices appear when we use these abilities in the word-based compressed text scenario that we have just described. Consider the automaton of Figure 4. It can search in the compressed text for a phrase of four words allowing up to two insertions, deletions or replacements of *words*. Apart from the well known horizontal transitions that match characters, there are vertical transitions that insert new words in the pattern, diagonal transitions that replace words, and dashed diagonal transitions delete words from the pattern.

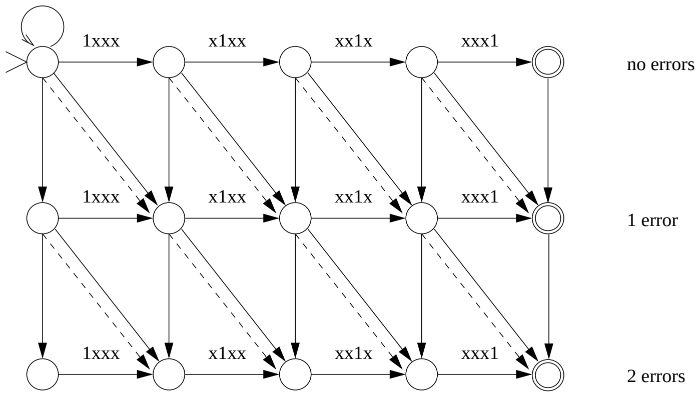


Figure 4: A nondeterministic automaton for approximate phrase searching (4 words, 2 errors) in the compressed text. Dashed transitions flow without consuming any text input. The other unlabeled transitions accept any bit mask.

This automaton can be efficiently simulated using extensions of the Shift-Or algorithm, so we can search in the compressed text for *approximate* occurrences of the phrase. For instance, the search for `"identifying potentially relevant matches"` could find the occurrence of `"identifying a number of relevant matches"` in the text with one replacement error, assuming that the stop words `"a"` and `"of"` are disregarded as explained before. Moreover, if we allow three errors at the character level as well we could find the occurrence of `"who identified a number of relevant matches"` in the text, since for the algorithm there is an occurrence of `"identifying"` in `"identified"`.

Other efficiently implementable setups can be insensitive to the order of the words in the phrase. The same phrase query could be found in `"matches considered potentially relevant were identified"` with one deletion error for `"considered"`. *Proximity searching* is also of interest in IR and can be efficiently solved. The goal is to give a phrase and find its words relatively close to each other in the text. This would permit to find out the occurrence of `"identifying and tagging potentially relevant matches"` in the text.

Approximate searching has traditionally operated at the character level, where it aims at recovering the correct *syntax* from typing or spelling mistakes, errors coming from optical character recognition software, misspelling of foreign names, and so on. Approximate searching at the word level, on the other hand, aims at recovering the correct *semantics* from concepts that are written with a different wording. This is quite usual in most languages and is a common factor that prevents finding the relevant documents.

This kind of search is very difficult for a sequential algorithm. Some indexed schemes permit proximity searching by operating on the list of exact word positions, but this is all. In the scheme described above, this is simple to program, elegant and extremely efficient (more than on characters). This is an exclusive feature of this compression method that opens new possibilities aimed at recovering the intended semantics, rather than the syntax, of the query. Such capability may improve the retrieval effectiveness of IR systems.

4 Compression Methods for Inverted Indexes

An IR system normally uses an inverted index to quickly find the occurrences of the words in the text. Until now we have only considered text compression, but the index can be compressed as well. An attractive property of word-based Huffman text compression is that it integrates very well with an inverted index [35]. Since the table of Huffman symbols is precisely the vocabulary of the text, the data structures of the compressor and the index can be integrated, which reduces space and I/O overhead. At search time, the inverted index searches the patterns in the vocabulary; while if a sequential scan is necessary the online algorithm also searches the pattern in the vocabulary. Both processes can therefore be merged. We now show how text and index compression can be combined.

Three different types of inverted indices can be identified. The first one, called "full inverted index", stores the exact positions of each word in the text. All the query processing can be done using the lists and the access to the text is not necessary at all. In this case, the text can be compressed with any method, since it will only be decompressed to be presented to the user.

The second type is the "inverted file", which stores the documents where each word appears. For single word queries it is not necessary to access the text, since 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. For these queries the index must search directly the text of those documents where all the relevant words appear. If the text is to be compressed, an efficient compression scheme like that explained in previous sections must be used to permit fast online searching.

The third type, called "block addressing index", is presented in detail in the next section.

4.1 Block Addressing Index

Block addressing was first proposed in a system called *Glimpse* [25]. 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. 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. Any query must be solved using online searching 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 type of filter also needs efficiently searchable compression techniques if the text is to be kept compressed. Figure 5 illustrates a block addressing index with r blocks of b words each (i.e. $n = rb$). The Figure shows the entries in the block index for the word "blue", which has occurred in the blocks 1, 3 and r.

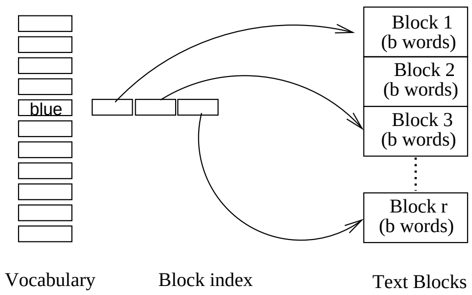


Figure 5: A block addressing index.

Searching in a block addressing index is similar to searching in a full inverted index. 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 [5], it may reduce space requirements with respect to using blocks of fixed size. Also, if we use blocks of fixed size and pack 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 [5], where an important result is analytically proved and experimentally verified: a block addressing index may yield sub-linear space overhead and at the same time sub-linear query time. Traditional inverted indexes pointing to words or documents achieve only the second goal. It is shown in [5] 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 sub-linear 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.

4.2 Combining Inverted Indexes and 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.

Any of the three schemes (full inverted index, inverted file, or block addressing index) can be combined with index compression. The size of an inverted index can be reduced by compressing the occurrence lists [45]. Because the list of occurrences within the inverted list is in ascending order, it can also be considered as a sequence of *gaps* between positions (be them text positions, document numbers, or block numbers). Since processing is usually done sequentially starting from the beginning of the list, the original positions can always be recomputed through sums of the gaps. By observing that these gaps are small for frequent words and large for infrequent words, compression can be obtained by encoding small values with shorter codes. Moreover, the lists may be built already in compressed form, which in practice expands the capacity of the main memory. This improves index construction times because the critical feature in this process is the amount of main memory available. In practice, text compression plus compressed index construction can be carried out faster than just index construction on uncompressed text.

The type of index chosen influences the degree of compression that can be obtained. The more fine-grained the addressing resolution of the occurrence lists (i.e. from pointing to blocks of text to pointing to exact positions), the less compression can be achieved. This occurs because the *gaps* between consecutive numbers of the occurrence lists are larger if the addressing is finer-grained. Therefore, the reduction in space from a finer-grained to a coarser-grained is more noticeable if index compression is incorporated into the system.

Comprehensive works showing how to compress inverted indexes can be found on the literature [24, 28, 45] 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 [14], 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 would be slower anyway. Further, the best parameterized

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

Table 4: Sample codes for integers.

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 [28, 45]. We present 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 4.

Elias-γ coding: Elias [10] 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 4.

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 4 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 [45], 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.

4.3 Improving the Index Compression

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

4.4 In-Memory Bucket Compression

In other compressed inverted schemes [45] the generation of the inverted list proceeds in a first step and their compression in a second step. 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. Since we already know the vocabulary we store a separate list of occurrences of each word and each new text occurrence is added at the end of the occurrence list of the corresponding word. Therefore, the gaps are computed and stored 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. Zipf's law states that, if we order the words of a natural language text in descending order of probability then the probability of the first word is i times the probability of the i-th word, for every i. 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 [17, 27, 15].

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 6 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 6. 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 6, these pointers were represented in 20 bits, allowing up to 2^{20} buckets in the main memory.

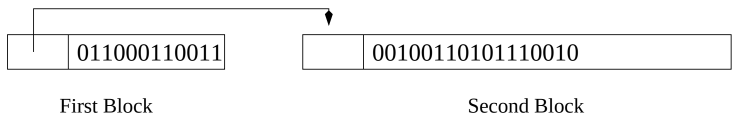


Figure 6: Linked list of buckets used with the in-memory compression scheme.

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 [28].

4.5 Compressing Frequencies

The research presented in [38] shows that for ranked queries it may be more efficient to store document numbers sorted by frequency rather than by document number. The compression techniques have to be adapted to this new problem. For instance, since the frequencies are decreasing, gaps between frequencies can be coded. Normally, in the list of each word there are a few documents with high frequencies and many with low frequencies. This information can be used to efficiently compress the frequencies. Moreover, all the documents with the same frequency can be stored in increasing document number, so gaps between documents can be used.

5 Integrated Approaches for IR Systems

We present in this section a combined design for block addressing index into a single approach, which includes text compression, block addressing and index compression. Figure 7 illustrates a block addressing structure, which has the following components:

Canonical Huffman Tree: the canonical Huffman tree is a small extra structure which is used to decode the text and to perform the sequential search on the blocks.

Vocabulary: the vocabulary of the collection is useful both as the symbol table of the Huffman coding in use and as the inverted index vocabulary.

Compressed Block Index: 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 4. Despite that separators are kept in the vocabulary for decompression purposes, there are no list of occurrences for them in the index. Another common choice in inverted indexes is to filter the words using techniques such as map letters to lowercase and apply stemming [6]). This choice may be done by not constructing lists of occurrences for these lists, as it is done with the separators. Figure 7 presents the block list for the word "blue".

Text Blocks: the text blocks form a logical layer over the natural document structure of the collection, so that the documents are not physically split or concatenated. This is implemented as a list of the documents of the collection, so that the position of a document 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 document identifiers, not necessarily matching the document boundaries. For each block we store the identifier of the last document that it spans in the list, and the offset of the last byte in that document that belongs to the block.

Compressed Text: each original document in the collection is compressed using a single canonical Huffman coding for the whole collection.

The space of this index is analyzed in [5], 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 documents are negligible in 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 [5], the β value for small blocks is larger than its asymptotic value, but it converges very close to it for relatively small blocks.

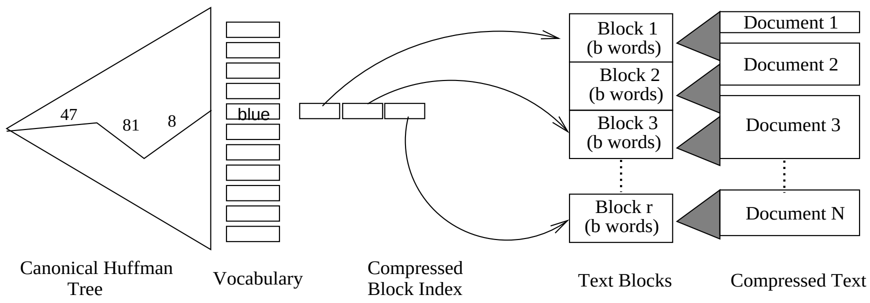


Figure 7: The structure of our index.

5.1 Index 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 steps.

Step 1

The first step corresponds to finding all the global information of interest. This step has the following main components: determine the set of documents 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 as the list of documents can be output to disk as they are found. At the end of this pass, we have computed the list of documents 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 step we run the Huffman algorithm on the vocabulary and build the canonical Huffman tree (see Section 2.1. 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 step. 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.

Step 2

The second step does the heavier part. Each text document of the collection is compressed, in the order dictated by the list of documents. Each word or separator of the text is searched in the vocabulary and its Huffman code is output to the compressed document. When we finish with a document, 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 4.4. 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 document being processed and store the number of bytes already written in the compressed version of the document. 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 document 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.

Step 3

At the end of Step 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.

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 [34]. The other processes of the first step are also of linear time and negligible in practice.

The second step 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 step 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 step 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 megabytes of RAM and a 500 words block length (a small block size), we are able to index a collection size close to 1 gigabytes without needing Step 3.

5.2 Query Searching

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

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, exactly or allowing errors. 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. 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. 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).

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 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 $\ell_1 \cap \ell_2 = \ell_2 - \ell_1^c$.

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 documents 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 multi-pattern 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 [5], but not tested.

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 loosing 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, can be accommodated in a very similar way. Proximity search corresponds to a sequence of ℓ words that must appear in correct order in an interval of ℓ' text words). Only the sequential searching changes, and the block boundaries are made overlap in $\ell' - 1$ words. Techniques to search diverse sophisticated patterns can be found in [33].

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 [5]. 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 [2].

However, phrase searching is much better. Using Zipf's Law, the shortest list among 2 or more random words has constant length, as shown in [2]. 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 Updating the Text Collection

An important issue regarding text databases is how to update the index when changes occur. Inverted indices normally handle those modifications by building differential inverted indices on the new or modified texts and then merging periodically the main and the differential index. When using compression, however, a new problem appears because the *frequencies* of the words change too, and therefore the current assignment of compressed codes to them may not be optimal anymore. Even worse, new words may appear which have no compressed code. The naive solution of recompressing the whole database according to the new frequencies is too expensive, and the problem of finding the cheapest modification to the current Huffman tree has not been yet solved.

Fortunately, simple heuristics work well. At construction time, a special empty word with frequency zero is added to the vocabulary. The code that this empty word receives is called the *escape* code and is used to signal new words that appear. When new text is added, the existing words use their current code, and new words are codified as the escape code followed by the new words in plain form. Only at long periods the database is recompressed to recover optimality. If the text already compressed is reasonably large, the current code assignments are likely to remain optimal, and the number of new words added pose negligible overheads. For instance, experiments have shown a degradation of only 1% in the compression ratio after adding 190 megabytes to a text database of 10 megabytes. The search algorithms have also to be adapted if these new words are to be searched.

7 Final Remarks

7.1 Trends and research issues

For effective operation in an IR environment a compression method should satisfy the following requirements: good compression ratio, fast coding, fast decoding, and direct searching without the need to decompress the text. A good compression ratio saves space in secondary storage and reduces communication costs. Fast coding reduces processing overhead due to the introduction of compression into the system. Sometimes, fast decoding is more important than fast coding, as it happens in documentation systems in which a document is compressed once and decompressed many times from disk. Fast random access allows efficient processing of multiple queries submitted by the users of the information system. Fast sequential search reduces query times in many types of indices.

We have shown compression techniques that satisfy all these goals. The compressed text *plus* a compressed inverted index built on it take no more than 40% of the original text size *without* any index. Index construction, *including* the text compression, proceeds *faster* than the indexing of the uncompressed text, and needs less space. Any search scheme, be it based on the index, on sequential searching, or on a combination of both, proceeds *faster* than on the uncompressed text and allows more flexible searching. Moreover, the text can be kept compressed *during all times*, being decompressed only to be displayed to the user. Such combination of features is unbeaten in efficiency and flexibility, and permits transparent integration into not only traditional text databases but also, for example, Web servers and compressed file systems, with the additional benefit of providing searchability [35, 50].

The most adequate indexing technique in practice is the inverted file. The main trends in indexing and searching textual databases are influenced by the following aspects. Searching is becoming more complex. As textual databases grow and become error-prone, enhanced query facilities are required, such as allowing errors in the text. At the same time, text collections are becoming huge, thus imposing more demanding requirements at all levels. With the speed of processors and the relative slowness of disks, it is better to keep the texts compressed, because reading less text from disk and decompressing it in main memory pays off. Also, techniques such as block addressing trade space for processor time. Considering the aspects just mentioned we might say that compression is becoming a very important issue for next generation text retrieval systems.

7.2 Bibliographic Discussion

Regarding text compression, several books are available. Most of the topics discussed here are covered in more detail by Witten, Moffat and Bell [45]. They also present implementations of text compression methods, such as Huffman and arithmetic coding. Bell, Cleary and Witten [7] cover statistical and dictionary methods, laying particular stress on adaptive methods as well as theoretical aspects of compression, with estimates on the entropy of several natural languages. Storer [41] covers the main compression techniques, with emphasis on dictionary methods. Moffat and Turpin [29] covers compression and coding algorithms.

Huffman coding was originally presented in [20]. Adaptive versions of Huffman coding appear in [13, 21, 42]. Word-based compression is considered in [9, 26, 19, 8]. Canonical codes were first presented in [40]. Many properties of the canonical codes are mentioned in [18]. Byte Huffman coding was proposed in [32]. Sequential searching on byte Huffman compressed text is described in [32, 31, 50].

Sequential searching on Ziv-Lempel compressed data is presented in [12, 1]. More recently, implementations of sequential searching on Ziv-Lempel compressed text are presented in [36].

A variety of compression methods for inverted lists are studied in [30]. The most effective compression methods for inverted lists are based on the sequence of gaps between document numbers, as considered in [8] and in [28]. Their results are based on run-length encodings proposed by Elias [10] and Golomb [14]. A comprehensive study of inverted file compression can be found in [45]. More recently Vo and Moffat [43] have presented algorithms to process the index with no need to fully decode the compressed index.

A detailed explanation of a full inverted index and its construction and querying can be found in [2]. This work also includes an analysis of the algorithms on inverted lists that take into consideration the distribution of words in natural language texts. An in-place construction of inverted files is described in [28].

The idea of block addressing inverted indexes was first presented in [25]. This work presented a system called Glimpse, which also first explored the idea of performing complex pattern matching using the vocabulary of the index. Block addressing indexes are analyzed in [5], where some performance improvements are proposed. One variant for block addressing that indexes sequences instead of words has been implemented in a system called Grampse, described in [23].

References

[1] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: pattern matching in z-compressed files. *Journal of Computer and Systems Sciences*, 52(2):299–307, 1996.

[2] M. D. Araújo, G. Navarro, and N. Ziviani. Large text searching allowing errors. In R. Baeza-Yates, editor, *Proc. of the 4th South American Workshop on String Processing*, volume 8, pages 2–20. Carleton University Press International Informatics Series, 1997.

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

[4] R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

[5] R. Baeza-Yates and G. Navarro. Block-addressing indices for approximate text retrieval. *Journal of the American Society for Information Science (JASIS)*, 51(1):69–82, January 2000.

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

[7] T. Bell, J. Cleary, and I. Witten. *Text Compression*. Prentice Hall, New Jersey, 1990.

[8] T. C. Bell, A. Moffat, C. Nevill-Manning, I. H. Witten, and J. Zobel. Data compression in full-text retrieval systems. *Journal of the American Society for Information Science*, 44:508–531, 1993.

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

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

[11] Google Search Engine. http://www.google.com.

[12] M. Farach and M. Thorup. String matching in lempel-ziv compressed strings. In *Proc. 27th ACM Annual Symposium on the Theory of Computing*, pages 703–712, 1995.

[13] R. G. Gallager. Variations on a theme by huffman. *IEEE Transactions on Information Theory*, 24:668–674, 1978.

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

[15] D. Harman, E. Fox, R. Baeza-Yates, and W. Lee. *Inverted files*, pages 28–43. Prentice-Hall, 1992.

[16] D. K. Harman. Overview of the third text retrieval conference. In *Proc. Third Text REtrieval Conference (TREC-3)*, pages 1–19, Gaithersburg, Maryland, 1995. National Institute of Standards and Technology Special Publication.

[17] D. Hawking. Scalable text retrieval for large digital libraries. In C. Peters and C. Thanos, editors, *Proc. of the 1st European Conference on Digital Libraries*, number 1234 in LNCS, pages 127–146, Pisa, Italy, September 1997. Springer-Verlag.

[18] D. S. Hirschberg and D. A. Lelewer. Efficient decoding of prefix codes. *Communications of the ACM*, 33(4), 1990.

[19] R. N. Horspool and G. V. Cormack. Constructing word-based text compression algorithms. In *Proc. of IEEE Second Data Compression Conference*, pages 62–81, 1992.

[20] D. Huffman. A method for the construction of minimum-redundancy codes. In *Proc. of the Institute of Electrical and Radio Engineers*, volume 40, pages 1090–1101, 1952.

[21] D. E. Knuth. Dynamic huffman coding. *Journal of Algorithms*, 6:163–180, 1985.

[22] S. Lawrence and C. L. Giles. Accessibility of information on the web. *Nature*, 400(8):107–109, 1999.

[23] O. Lehtinen, E. Sutinen, and J. Tarhio. Experiments on block indexing. In *Proc. of the 3th South American Workshop on String Processing (WSP'96)*, pages 183–193. Carleton University Press, 1996.

[24] G. Linoff and C. Stanfill. 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)*, pages 88–95, 1993.

[25] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proc. USENIX Technical Conference*, pages 23–32. USENIX Association, Berkeley, CA, USA, Winter 1994.

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

[27] A. Moffat. Economical inversion of large text files. *Computing Systems (USENIX Assoc. Journal)*, 5(2):125–139, 1992.

[28] A. Moffat and T.A.H. Bell. In-situ generation of compressed inverted files. *Journal of the American Society for Information Science*, 46(7):537–550, August 1995.

[29] A. Moffat and A. Turpin. *Compression and Coding Algorithms*. Kluwer Academic Publishers, 2002.

[30] A. Moffat and J. Zobel. Parameterized compression for sparse bitmaps. In *Proc. of the ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 274–285, 1992.

[31] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Direct pattern matching on compressed text. In *Proc. of the 5th Symposium on String Processing and Information Retrieval (SPIRE'98)*, pages 90–95. IEEE CS Press, 1998. `ftp://ftp.dcc.uchile.cl/-pub/users/gnavarro/spire98.3.ps.gz`.

[32] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast searching on compressed text allowing errors. In *Proc. of the 21st International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 298–306, 1998.

[33] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139, 2000.

[34] E. S. Moura, G. Navarro, and N. Ziviani. Linear time sorting of skewed distributions. In *Proc. 6th String Processing and Information Retrieval Symposium (SPIRE'99)*, pages 135–140, Cancun, Mexico, 1999. IEEE Computer Society.

[35] G. Navarro, E. Moura M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Information Retrieval*, 3(1):49–77, 2000.

[36] G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In *Proc. of the 9th Symposium on Combinatorial Pattern Matching (CPM'98)*, number 1448 in LNCS, pages 14–33. Springer-Verlag, 1998.

[37] D. A. Patterson and J. L. Hennessy. *Computer architecture: A quantitative approach.* Morgan Kaufmann Publishers, 2nd edition, 1995.

[38] M. Persin. Document filtering for fast ranking. In *Proc. of the 17th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 339–348. Springer Verlag, July 1994.

[39] J. Rissanen and G. G. Langdon. Arithmetic coding. *IBM Journal of Research and Development*, 23:149–162, 1979.

[40] E. S. Schwartz and B. Kallick. Generating a canonical prefix encoding. *Communications of the ACM*, 7:166–169, 1964.

[41] J. A. Storer. *Data Compression: Methods and Theory.* Computer Science Press, 1988.

[42] J. S. Vitter. Algorithm 673: Dynamic huffman coding. *ACM Transactions on Mathematical Software*, 15:158–167, 1989.

[43] A. N. Vo and A. Moffat. Compressed inverted files with reduced decoding overhead. In *Proc. of the ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 290–297, 1998.

[44] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[45] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes.* Morgan Kaufmann Publishers, second edition, 1999.

[46] I. H. Witten, R. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–541, 1987.

[47] S. Wu and U. Manber. Fast text searching allowing errors. *Comm. of the ACM*, 35(10):83–91, October 1992.

[48] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[49] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

[50] N. Ziviani, E. Moura, G. Navarro, and R. Baeza-Yates. Compression: A key for next-generation text retrieval systems. *IEEE Computer*, 33(11):37–44, November 2000.