Compression: A Key for Next Generation Text Retrieval Systems*

Nivio Ziviani Edleno Silva de Moura

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

Gonzalo Navarro Ricardo Baeza-Yates

Department of Computer Science
Univ. de Chile, Chile

September 27, 2000

Abstract

In this article 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 one of the few cases 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.

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 [3, 11]. In [6], 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. 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 article 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. To illustrate this point consider the problem of creating a large text database and providing fast access through keyword searches. By compressing both the index and the complete text, 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

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

and text had not been compressed. This is one of the few cases where there is no space-time trade-off.

Traditionally, compression techniques have not been used in IR systems because the compressed texts did not allow fast access. Recent text compression methods [7] 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 the 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. From 1980 to 1995 the time to access disks kept approximately constant while processing speed increased approximately 2,000 times [9]. 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 much less 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 [3, 11]. 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 this article, we first review traditional and recent methods for compressing text. Following, we discuss how recent techniques permit fast direct searching on the compressed text.

Later on, we show how the new techniques can improve the overall efficiency of IR systems. Moreover, the text can be kept compressed all the time, allowing updates when changes occur in the compressed text.

Text Compression Methods for IR Systems

One well-known coding strategy is Huffman coding [5]. 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. New 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.

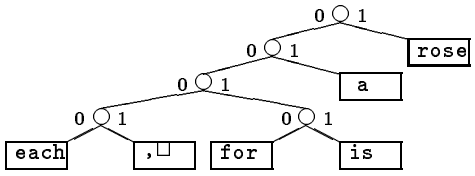
Another important family of compression methods, called Ziv-Lempel, substitutes a sequence of symbols by a pointer to a previous occurrence of that sequence. Compression is obtained because the pointers need less space than the phrase they replace. Ziv-Lempel methods are popular for their speed and economy of memory, 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 [11], another well-known compression method, presents similar problems.

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 separators. An efficient way to deal with words and separators is to use a method called *spaceless words* [7]. 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.

The example also shows how the codes for the symbols are organized in a so-called *Huffman tree*. The most frequent word (in this case, `"rose"`) receives the shortest code (in this case,



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.

"1"). The Huffman method gives the tree that minimizes the length of the compressed file, but many trees would have achieved the same compression. For instance, exchanging the left and right children of a node yields an alternative Huffman tree with the same compression ratio. The preferred choice for most applications is the *canonical tree*, where the right subtree is never taller than the left subtree, as it happens in Figure 1. Canonical trees allow more efficiency at decoding time with less memory requirement. The algorithm for building the Huffman tree from the symbol frequencies is described, for instance, in [11], and can be done in linear time after sorting the symbol frequencies.

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".

Byte-Oriented Huffman Coding

The original method proposed by Huffman is mostly used as a binary code. In [7], 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 article. 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 "rose" could be the 3-byte code "47 31 8", and a possible tagged code for the word "rose" could be "175 31 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 1 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 [4]. 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.

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 1: Comparison of compression techniques on the WSJ text collection.

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. The exact search can be done on the compressed text directly, using any known sequential pattern matching algorithm. The general algorithm 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 search. As we see later, it can also be used to improve indexed schemes that combine inverted files and sequential search [8, 11].

Online Searching on 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 [7].

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 multipattern search for all the codes is then conducted on the text [7]. 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.

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. We present a more general approach now, which works also on plain Huffman codes. We start with the search algorithm for simple words and then show progressively how the query can be extended to phrases formed by complex patterns, keeping the simplicity of the approach.

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 `"rose"`, encoded as the 3-byte `47 31 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.

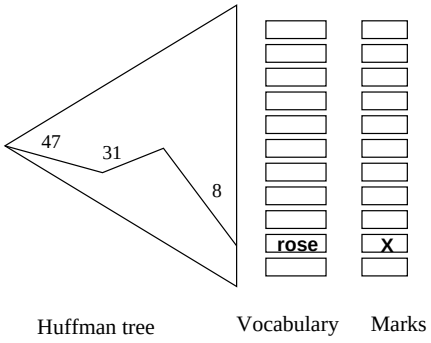


Figure 2: General searching scheme for the word `"rose"`.

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 "ro* rose is" allowing one 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, as the mask is "110" for this 3-elements phrase.

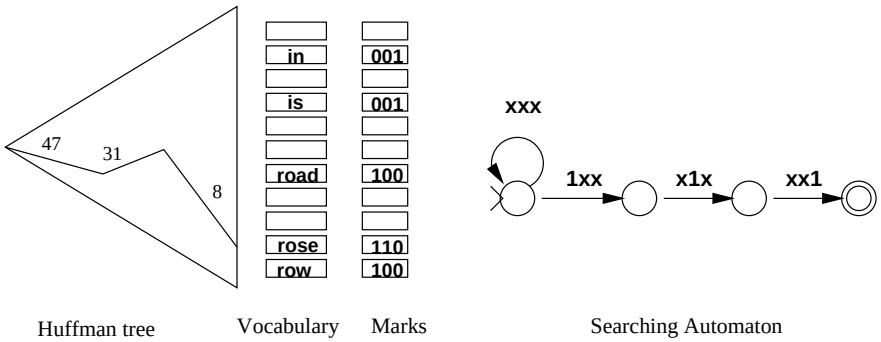


Figure 3: General searching scheme for the phrase "ro* rose is" allowing 1 error. The x in the automaton stands for 0 or 1.

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 "ro* rose 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 [1]. 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 text character. 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 text character, 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 2 presents exact ($k = 0$) and approximate ($k = 1, 2, 3$) searching times for the WSJ file using *Agrep* [12], 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 2: 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 [12, 2]. This powerful type of search is the basis of the software Agrep [12].

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.

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

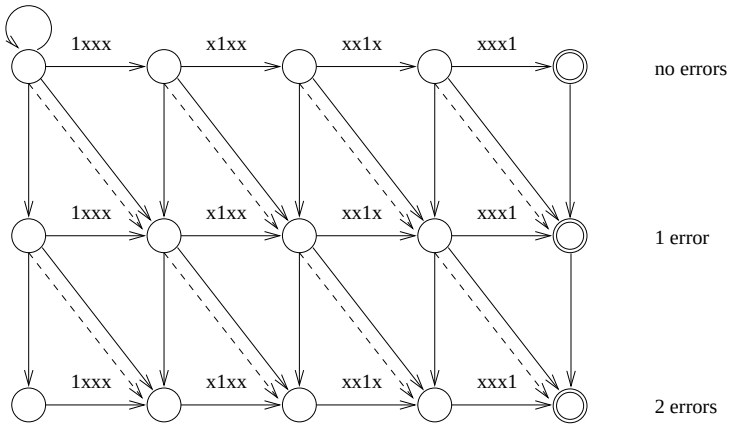


Figure 4: A nondeterministic automaton for approximate phrase searching (4 words, 2 errors) in the compressed text. Dashed transitions flow without consuming any text input. The other unlabeled transitions accept any bit mask.

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.

Using Compression on Inverted Indices

An IR system normally uses an inverted index to quickly find the occurrences of the words in the text. Up to now we have only considered text compression, but the index can be compressed as well. 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", 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. The space occupied by the index may be small, but almost 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.

Any of the above schemes can be combined with index compression. The size of an inverted index can be reduced by compressing the occurrence lists [11]. 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.

An attractive property of word-based Huffman text compression is that it integrates very well with an inverted index [8]. 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.

The research presented in [10] 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.

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.

Conclusions

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 new 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 filesystems, with the additional benefit of providing searchability [8].

Acknowledgements

We wish to acknowledge the many comments of Berthier Ribeiro-Neto and Wagner Meira Jr that helped us to improve this article.

References

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

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

[3] R. Baeza-Yates and B. Ribeiro-Neto, editors. *Modern Information Retrieval.* Addison-Wesley, 1999. 513 pages.

[4] 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 Publica tion.

[5] 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.

[6] S. Lawrence and C. Giles. Accessibility of information on the web. *Nature*, pages 107–109, August 1999.

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

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

[9] D. Patterson and J. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Francisco, 2nd edition, 1995.

[10] 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.

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

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