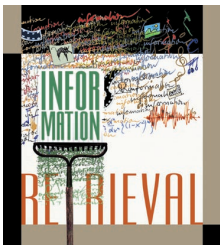
Compression: A Key for Next-Generation Text Retrieval Systems



The continually growing Web challenges information retrieval systems to deliver data quickly. The authors' technique combines several data compression features to provide economical storage, faster indexing, and accelerated searches.

Nivio Ziviani
Federal University of Minas Gerais, Brazil

Edleno Silva de Moura
Federal University of Amazonas, Brazil

Gonzalo Navarro

Ricardo Baeza-Yates
University of Chile

As online textual information explodes through the widespread use of digital libraries, office automation systems, document databases, and the Web, the need arises for an effective information retrieval (IR) system.[1,2] The Web alone comprises approximately 800 million static pages, containing 6 trillion bytes of plain text—enough to store the text of a million books.[3] Because text retrieval is the kernel of most IR systems, today's IR systems face the dynamic challenge of providing rapid and immediate access to this textual mass.

In this article, we discuss the recent techniques that permit a fast and direct method for searching compressed text, and we explain how these new techniques can improve the overall efficiency of IR systems.

RAPID, EFFICIENT TEXT RETRIEVAL

Consider the problem of creating a large text database and providing fast access through keyword searches. Compressing both the index and the complete text cuts the total space in half. The time required to build the index and answer a query is far less than if the index and text had not been compressed. This illustrates a rare case where there is no space-time trade-off.

Traditionally, IR systems have not used compression techniques, because the compressed texts did not allow rapid access. However, recent methods have demonstrated that directly searching the compressed text is faster than searching the original text and that flexible word searching improves the amount of compression obtained, as explained in our recent work.[4] Direct access to any point in the compressed text is also possible. For example, the IR system can access a given word in a compressed text without decoding the entire text from the beginning. These new features and advantages of text compression lead to a win-win situation that regenerates interest in text compression for IR systems.

Text compression focuses on finding ways to represent actual text in less space. This process involves replacing text symbols with equivalent ones that use a smaller number of bits or bytes. In addition to being cost-efficient for large text collections, text compression is attractive because it

- requires less storage space,
- speeds up disk reads and data transmittals, and
- reduces search time.

The cost of text compression is mainly due to text coding and decoding. This cost, however, is becoming less significant as technology progresses. In the past 20 years, the time to access disks remained almost constant, whereas the processing speed increased approximately 2,000 times.[5]

The compression ratio—the compressed file's size as a percentage of the uncompressed file's size—quantifies the amount of space savings. Other important parameters include compression and decompression speed. In some situations, decompression speed is more important than compression speed, such as in textual

0018-9162/00/$10.00 © 2000 IEEE

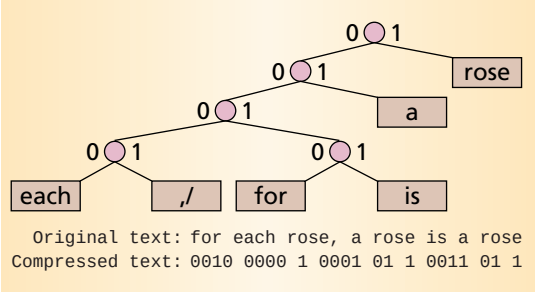


Figure 1. Compression using Huffman coding for the spaceless-words method. The example illustrates the organization of the set of symbols in a Huffman tree. When a word is followed by a space, the word is encoded; otherwise, the word and the separator are encoded.

databases and documentation systems, in which compressing the text once and then reading it many times from a disk are common.

Searching compressed text

Another important technique is using pattern matching on a compressed text without decompressing it. Compressing the search key rather than decoding the compressed text can accelerate sequential searching. Consequently, using compressed text makes the search faster because fewer bytes must be scanned.

Indexing

Efficient text retrieval from large text collections requires specialized indexing techniques. An index is a data structure built on the text collection to speed up queries. A simple and popular indexing structure for text collections is the inverted file,[1,2] which is adequate when the pattern to be searched contains simple words—for example, Web pages that include the words "text" and "compression." An inverted file typically includes the vocabulary—a vector containing all distinct words in the text collection—and a list of all document numbers in which each distinct word occurs; sometimes, the word frequency is also stored. The lists of document numbers constitute the largest part of an index. Researchers have proposed specific compression methods with good compression ratios to handle these lists. In this case, index compression schemes can significantly improve both index construction time and query processing time.

TEXT COMPRESSION METHODS FOR IR SYSTEMS

There are various coding strategies; however, not all coding strategies are suitable for information retrieval systems. For example, Ziv-Lempel, an important family of compression methods, replaces a sequence of symbols with a pointer to a previous occurrence of that sequence. Compression occurs because the pointers need less space than the phrases they replace.

Although Ziv-Lempel methods are popular because of their speed and economy of memory, these methods present important disadvantages in an IR environment. First, they require decoding to start at the beginning of a compressed file, making random access expensive. Second, the compressed files are difficult to search without decompressing the files. A possible advantage is that they do not need to store a table of symbols as some other compression methods do, but this bears little importance in IR scenarios because the vocabulary of the text is required for indexing and querying purposes. Arithmetic coding,[2] another well-known compression method, presents similar problems.

One well-known coding strategy is Huffman coding,[6] where the concept is to compress the text by assigning shorter codes to symbols with higher frequencies. This coding involves assigning a unique variable-length bit encoding to each different text symbol. The traditional implementations of the Huffman method are character based; that is, certain characters act as the symbols in the alphabet. To tailor compression algorithms to IR systems, we treat the symbols as compressed words instead of characters.

Word-based Huffman compression

For natural language texts used in an IR context, the most effective compression technique is word-based Huffman coding. Words are the atoms on which most IR systems are built. By translating words as symbols, the table of symbols in the compression coder becomes the text vocabulary, hence 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—a critical issue for IR systems. Moreover, character-based Huffman methods can typically compress English texts by about 40 percent, whereas word-based Huffman methods can reduce them by more than 75 percent, because the word distribution is more biased than the character distribution.

Compression occurs 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 performs the actual compression during a second pass.

The text consists not only of words but also of separators. An efficient way to handle words and separators is to use *spaceless words*, a method that we proposed in our recent work.[4] When a single space follows a word, the compression algorithm encodes only the word; otherwise, the algorithm encodes the word, then the separator. During decoding, the compression algorithm treats each word as if a space fol-

lowed it, except when the next symbol corresponds to a separator. Figure 1 illustrates compression using Huffman coding for the spaceless-words method. The set of symbols in this case is {"a, ""each," "is," "for, " "rose," "*,/*" (where / represents space), whose frequencies are 2, 1, 1, 1, 3, 1, respectively.

In Figure 1, a Huffman tree organizes the symbols. The most frequent word (in this case, "rose") receives the shortest code (in this case, "1"). According to the Huffman method, the tree minimizes the length of the compressed file; however, many trees would have achieved the same compression. For example, 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 one. Canonical trees allow more efficiency at decoding time with less memory requirement. For instance, Ian Witten and colleagues[2] described an algorithm for building the Huffman tree from the symbol frequencies. Such an algorithm can run in linear time if the symbols are already sorted by their frequencies.

To decompress text, the algorithm sequentially traverses the stream of bits in the compressed file. Starting at the root, the algorithm uses the sequence of bits read to traverse the Huffman tree. When the algorithm reaches a leaf node, it prints the corresponding word—the decompressed symbol—and restarts the tree traversal. Thus, for the tree in Figure 1, the code "0010" in the compressed file leads to the decompressed symbol "for."

Byte-oriented Huffman coding

The original method that Huffman proposed is mostly used as a binary code. We use the Huffman algorithm in another way, modifying the code assignment to associate a sequence of whole bytes with each word in the text.[4] As a result, the maximum degree of each node is now 256. In this article, we refer to this version as *plain Huffman code*. An alternative use of byte coding, called *tagged Huffman code*, uses only seven of the eight bits of each byte for the code; hence, the tree has degree 128. The eighth bit signals the first byte of each code word, which 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 no significant degradation of the compression ratio due to 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. Comparison of compression techniques on *The Wall Street Journal* text collection.

Method	Compression ratio (percentage)	Compression time (minutes)	Decompression time (minutes)
Binary Huffman	27.13	8.77	3.08
Plain Huffman	30.60	8.67	1.95
Tagged Huffman	33.70	8.90	2.02
Gnu Gzip	37.53	25.43	2.68
Unix Compress	42.94	7.60	6.78

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 WSJ file, which contains all *The Wall Street Journal* issues from 1987, 1988, and 1989—part of the TREC 3 collection.[7] The WSJ file has 250 megabytes, almost 43 million total words, and nearly 200,000 different words (vocabulary). Using bytes rather than bits in the Huffman code degrades the compression ratio only slightly. The increase in the compression ratio of the tagged Huffman code is approximately three points more than that of the plain Huffman code. This difference is due to the extra space allocated for the tag bit in each byte. The compression time using bytes is two to three times faster than gnu Gzip and only 17 percent slower than Unix Compress, which achieves poor compression ratios. The decompression time using bytes, in both tagged and plain Huffman, is more than 20 percent faster than gnu Gzip and three times faster than Unix Compress.

One important advantage in using byte-oriented Huffman coding is the possibility of performing a fast, direct search of compressed text. You can perform the exact search directly on the compressed text using any sequential pattern-matching algorithm. The general algorithm allows large variations of exact and approximate searching, such as phrases, ranges, complements, wild cards, and arbitrary regular expressions. This technique is useful not only for speeding up sequential search but also for improving indexed schemes that combine inverted files and sequential search.[2,8]

ONLINE SEARCH OF COMPRESSED TEXT

One of the most attractive properties of the byte-oriented Huffman method is that it lets you search compressed text exactly as you would search uncompressed text. When you submit a query, the text is in compressed form, and the pattern is in uncompressed form. An essential step in this direct search technique is to compress the pattern rather than uncompress the text.

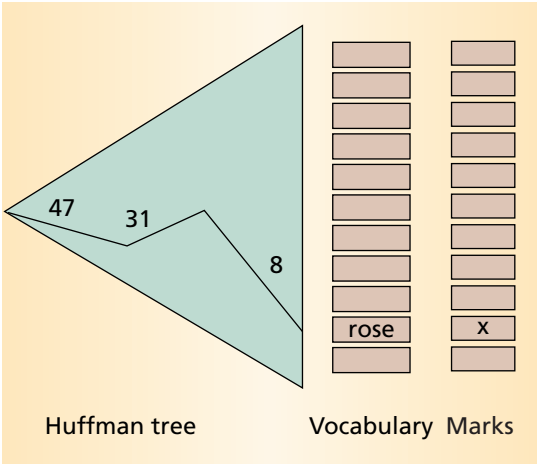


Figure 2. General searching scheme for the word "rose." The algorithm for the pattern "rose" is encoded as 47 31 8 in 3 bytes. The algorithm reports an occurrence when it finds this sequence of bytes in the compressed text and reaches the corresponding leaf in the Huffman tree.

The algorithm to find the single-word occurrences starts by searching for the word in the vocabulary, where binary searching is a simple, inexpensive choice. After finding the word, you also have the word's compressed code. Then, to search this compressed code in the text, you can use any classical string-matching algorithm with no modifications. This flexibility 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 although the word does not appear in the original text. This may happen in plain Huffman codes because the concatenation of the codes for other words may contain the code sought, but it is impossible in tagged Huffman code.[4]

Today's IR systems require flexibility in the search patterns. "Complex" patterns range from disregarding the upper- or lowercase letters 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, the IR system performs a sequential search in the vocabulary and collects the compressed codes of all words that match the pattern. The system then conducts a multipattern search for all codes on the text, as we previously described.[4] Sequential vocabulary searching is not expensive for natural-language texts because the vocabulary is small compared with the entire text (0.5 percent is typical for large texts). On the other hand, this sequential searching permits extra flexibility, such as allowing errors.

Flexible pattern matching

Although direct searching is efficient, extending it to handle complex queries is difficult. These complex queries consist of either phrases of complex patterns that an algorithm must search while allowing errors or regular expressions, or both. Our more general approach also works on plain Huffman codes. We start with the search algorithm for simple words, then we show progressively how we can extend the query to phrases formed by complex patterns while retaining the approach's simplicity.

The search algorithm for a single word starts again in the vocabulary using binary search. Once the algorithm finds the word, it marks the corresponding leaf in the Huffman tree. Next, it scans the compressed text, byte by byte. At the same time, the algorithm traverses the Huffman tree downward, as though it were decompressing the text but without generating it. Each time the algorithm reaches a leaf of the Huffman tree, a complete word has been read. If the leaf has a mark, the algorithm reports an occurrence. Despite the marking of the leaf, the algorithm returns to the root of the Huffman tree and resumes scanning the text. Figure 2 illustrates the algorithm for the pattern "rose," encoded as the 3-byte 47 31 8 sequence. Each time the algorithm finds this sequence of bytes in the compressed text, it reaches the corresponding leaf in the Huffman tree and reports an occurrence. Sequential searches in the vocabulary handle complex patterns. This time the algorithm marks all leaves corresponding with their matching words.

We can extend this simple scheme to handle complex *phrase queries*—sequences of patterns, each of which can be derived from a simple word or 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 ith bit of word x is set if x matches the ith element of the phrase query. The algorithm searches each element i of the phrase, in turn, in the vocabulary and marks the ith bit of the words that the element i matches. Figure 3 illustrates the masks for the pattern "ro* rose is," with this search scheme allowing one error per word, where "ro*" means any word starting with "ro." For example, the word "rose" in the vocabulary matches the pattern in positions 1 and 2, as the mask is "110" for the three-element phrase "ro* rose is."

After the preprocessing phase, the algorithm scans the text as before. A nondeterministic automaton of $i + 1$ states controls the search state for the pattern "ro* rose is" (Figure 3). The automaton moves from state i to state $i + 1$ whenever it recognizes the ith pattern of the phrase. State zero is always active, and the algo-

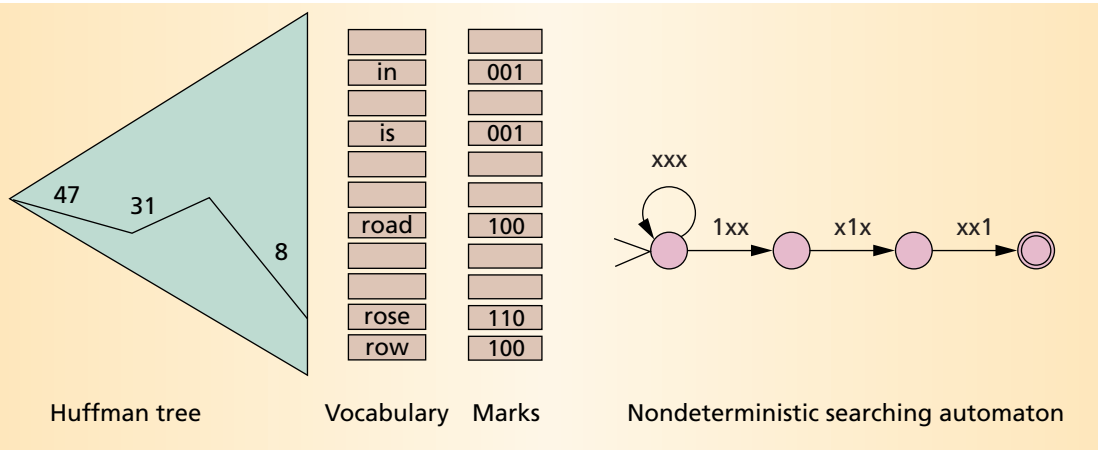


Figure 3. General searching scheme for the three-element phrase "ro∗ rose is," allowing one error per word. The "ro*" means any word that starts with "ro." The x in the automaton stands for 0 or 1.

rithm reports occurrences whenever state i is activated. The automaton is nondeterministic because, at a given moment, many states may be active. The algorithm reads the bytes of the compressed text and traverses the Huffman tree. Each time the algorithm reaches a leaf of the tree, it sends that leaf's bit mask to the automaton. An active state $i - 1$ will activate the state i only if the ith bit of the mask is active. Therefore, the automaton makes one transition per word of the text.

A Shift-Or algorithm can efficiently implement this automaton, as our earlier publication explains.[9] This algorithm can simulate an automaton of up to $w + 1$ states (where w is the length in bits of the computer word), performing two operations per text character. 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 require more machine words for the simulation, although the technique is the same.

The Shift-Or algorithm maps each automaton state, 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 the algorithm simulates using a *shift* in the bit mask. Only those states that match the current text word can actually pass; a bitwise *and* operation with the bit mask found in the leaf of the Huffman tree simulates the move. Therefore, one *shift* and one *and* operation per text word update the search state. The original Shift-Or algorithm uses the reverse bits for efficiency—hence, the name Shift-Or.

The algorithm can disregard the separators in the search, finding a phrase query even when there are two spaces instead of one. It can also disregard stop words (articles, prepositions, and so forth). The procedure is to ignore the corresponding leaves of the Huffman tree when the search reaches them. This ability is common in inverted files but 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,[10] the direct search on tagged Huffman, and the automaton search using plain Huffman. As the table reveals, both direct and automaton search algorithms are almost insensitive to the number of errors allowed in the pattern, whereas Agrep is not. Table 2 also shows that both compressed search algorithms are faster than Agrep—up to 50 percent faster for exact searching and nearly eight times faster for approximate searching. Automaton searching permits complex phrase searching for the same cost, and it allows more sophisticated searching. However, because automaton searching is always slower than direct searching, you should only use it for complex queries.

Table 2. Searching times (in seconds) for *The Wall Street Journal* text file, with 99 percent confidence.

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

Mean ± standard deviation values are expressed.

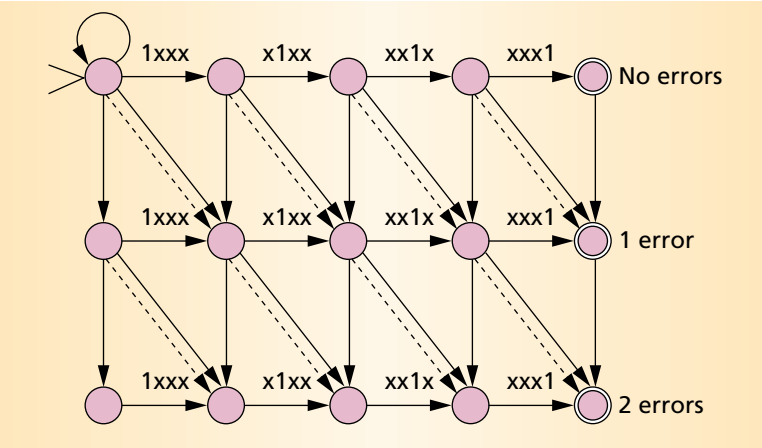


Figure 4. A nondeterministic automaton for approximate phrase searching (four words, two errors) in the compressed text. Dashed transitions flow without consuming any text input. The other unlabeled transitions accept any bit mask. You can use extensions of the Shift-Or algorithm to simulate this automaton. The objective of proximity searching is to give a phrase and locate its words close to one another in the text.

Enhanced searching

The Shift-Or algorithm—which is the basis for the Agrep software—is more than just a search for a simple sequence of elements; it can search for regular expressions and allow errors in the matches and in other flexible patterns.[10,11]

New choices appear when you use these abilities in a word-based compressed-text scenario. The automaton illustrated in Figure 4 can search the compressed text for a four-word phrase, allowing up to two insertions, deletions, or word replacements. In addition to the well-known horizontal transitions that match characters, vertical transitions insert new words in the pattern, diagonal transitions replace words, and dashed diagonal transitions delete words from the pattern.

We can efficiently simulate this automaton by using extensions of the Shift-Or algorithm so that we can search in the compressed text for approximate occurrences of the phrase. For example, the search for "identifying potentially relevant matches" can locate the occurrence of "identifying a number of relevant matches" in the text with one replacement error, assuming that it disregards the stop words "a" and "of." Moreover, if we allow three errors at the character level, we can locate the occurrence of "who identified a number of relevant matches" in the text because the algorithm recognizes an occurrence of "identifying" in "identified."

Other easy-to-implement setups can be insensitive to the order of the words in the phrase. The phrase "matches considered potentially relevant were identified" with one deletion error for "considered" contains the same phrase query. Proximity searching is also of interest in IR and can be efficiently solved. The goal of proximity searching is to give a phrase and find its words relatively close to one another in the text—permitting, for example, "identifying and tagging potentially relevant matches."

Approximate searching has traditionally operated at the character level, where it aims at recovering the correct syntax from typing or spelling mistakes, errors from optical character-recognition software, misspellings of foreign names, and so on. Approximate searching at the word level, on the other hand, aims at recovering the correct semantics from concepts written with different wording. Although typical of most languages, this common factor prevents finding the relevant documents.

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

USING COMPRESSION ON INVERTED INDICES

An IR system normally uses an inverted index to quickly find word occurrences in the text. Until now, we have considered only text compression, but we can also compress the index. We can identify three different types of inverted indices. The first, *full inverted index*, stores the exact positions of each word in the text. Because the query processing can use the lists, access to the text is unnecessary. We can compress the text using any method because decompression is reserved only for presentation to the user.

The second type, the *inverted file index*, stores the documents where each word appears. For single-word queries, accessing the text is not necessary because when a word appears in a document, the algorithm retrieves the whole document. However, phrase or proximity queries cannot be solved with the information that the index stores. Two words can appear in the same document without forming a phrase. For these queries, the index must directly search the text of those documents where all relevant words appear. If text compression is necessary, only an efficient compression scheme can permit fast online searching.

The third type, *block addressing index*, divides the text in blocks of fixed size, which can 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 most queries must be solved

using online searching because the location of the block where the word appears in the document is not known. The index serves to filter out some blocks of the collection that cannot contain a match. This filter also needs efficiently searchable compression techniques when the text must be compressed.

Combining inverted indices and index compression

We can combine any of the three types of inverted indices with index compression. Compressing the occurrence lists reduces the size of an inverted index.[2] Because the list of occurrences within the inverted list is in ascending order, we can also consider it as a sequence of gaps between positions (text positions, document numbers, or block numbers). As search systems usually process the inverted lists sequentially from the beginning of each list, we can always recompute the original positions using the sums of the gaps. These gaps are small for frequent words and large for infrequent words. Therefore, the algorithm can achieve compression by encoding small values with shorter codes.

The index construction algorithm can build the lists already in compressed form, making better use of the main memory's capacity of the computer system. This improves index construction times because the critical feature in this process is the amount of main memory available in the computer system. Text compression plus compressed index construction is faster than only index construction on uncompressed text.

The type of index chosen influences the degree of compression. The more fine-grained the addressing resolution of the occurrence lists—from pointing to blocks of text to pointing to exact positions—the less compression can be achieved because the gaps between the consecutive numbers of the occurrence lists are larger. Therefore, incorporating index compression in the system increases the space reduction from finer-grained to coarser-grained addressing.

An attractive property of word-based Huffman text compression is that it integrates well with an inverted index, as elaborated in our recent study.[8] Because the table of Huffman symbols is precisely the vocabulary of the text, we can integrate the data structures of the compressor and the index, thereby reducing space and I/O overhead. At search time, the inverted index searches the patterns in the vocabulary; when a sequential scan is necessary, the online algorithm also searches the pattern in the vocabulary. Therefore, we can merge both processes.

Compressing frequencies

Michael Persin's[12] research demonstrates that for ranked queries, storing document numbers by frequency may be more efficient than storing them by document numbers. Researchers must adapt compression techniques to this new problem. For example, because the frequencies are decreasing, we can code the gaps between frequencies. Normally, each word's list includes a few documents with high frequencies and many documents with low frequencies. The compression algorithm can use this information to efficiently compress the frequencies. Moreover, the compression algorithm can store all documents with the same frequency in increasing document numbers, using the gaps between the different documents.

An ideal compression method should have a good compression ratio, fast coding, rapid decoding, and direct searching without decompressing the text.

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 periodically merging the main and the differential indices. Using compression, however, introduces a new problem because the word frequencies change; therefore, the current compressed code assignments may no longer be optimal, and new words may appear that have no compressed codes. The naive solution of recompressing the entire database according to the new frequencies is expensive, and no one has yet found the least expensive modification to the current Huffman tree.

Fortunately, simple heuristics work well. At construction time, we add a special empty word with a frequency of zero to the vocabulary. The code for this empty word—the *escape code*—signals new words that appear. When we add new text, the existing words use their current codes, and the compression algorithm codifies new words as the escape code followed by the new words in plain form. During long searches, recompressing the database keeps the compression ratio at efficient levels. When the compressed text is reasonably large, the current code assignments may remain optimal, and the new words added may pose negligible overheads. For example, a degradation of only 1 percent occurs in the compression ratio after adding 190 Mbytes of the WSJ file to a compressed 10 Mbytes portion of the same WSJ file. We should adapt the search algorithm when it searches these new words.

To operate effectively in IR environments, an ideal compression method should have a good compression ratio, fast coding, rapid decoding, and direct searching without decompressing the text. A good compression ratio saves space in secondary storage and reduces communication costs. Fast coding reduces the processing overhead cost by introducing compression into the IR system. Sometimes, rapid decoding is more important than fast coding, as in

documentation systems when a document is compressed once and decompressed many times from a disk. Fast random access allows the efficient handling of multiple queries submitted by information system users. Fast sequential searches reduce query times in many indices.

Our compression techniques satisfy all of these requisites. The compressed text, plus a compressed inverted index built on it, occupies no more than 40 percent of the original text size without any index. Index construction, including text compression, occurs faster than the indexing of the uncompressed text and requires less space. Any search scheme—based either on the index, sequential searching, or a combination of both—proceeds faster than if the search system were based on uncompressed text; it also permits more flexible searching. The text can be kept compressed, and the user can easily decompress it if a textual display is preferred. Such combined features offer the advantages of efficiency and flexibility. Critical to future IR systems, these dynamic compression methods permit optimal integration of massive data into traditional text databases, Web servers, and compressed file systems, with a bonus of enhanced searchability for effective, instantaneous retrieval. ✷

Acknowledgments

We thank Berthier Ribeiro-Neto and Wagner Meira Jr. for their constructive and useful comments. This work was supported, in part, by the SIAM/DCC/UFMG project, from the Brazilian MCT/FINEP/PRONEX, grant 76.97.1016.00, the AMYRI project, from Spanish Agency Cyted, Brazilian Agency CNPq, grant 520916/94-8, and Chilean Agency CONICYT, grant 1990627.

References

1. R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval,* Addison Wesley Longman, Reading, Mass., 1999.
2. I. Witten, A. Moffat, and T. Bell, *Managing Gigabytes,* 2nd ed., Morgan Kaufmann, San Francisco, 1999.
3. S. Lawrence and C.L. Giles, "Accessibility of Information on the Web," *Nature,* July 1999, pp. 107-109.
4. E. Moura et al., "Fast and Flexible Word Searching on Compressed Text," *ACM Trans Information Systems*, Vol. 18, No. 2, 2000, pp. 113-139.
5. D.A. Patterson and J.L. Hennessy, *Computer Architecture: A Quantitative Approach*, 2nd ed., Morgan Kaufmann, San Francisco, 1995.
6. D. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proc. Inst. of Electrical and Radio Engineers*, Vol. 40, No. 9, 1952, pp. 1090-1101.
7. D.K. Harman, "Overview of the Third Text Retrieval Conference," *Proc. Third Text Retrieval Conference, TREC-3*, Int'l Inst. of Standards and Technology, Gaithersburg, Md., 1995, pp. 1-19.
8. G. Navarro et al., "Adding Compression to Block Addressing Inverted Indexes," *Information Retrieval,* Vol. 3, No. 1, 2000, pp. 49-77.
9. R. Baeza-Yates and G. Gonnet, "A New Approach to Text Searching," *Comm. ACM*, Oct. 1992, pp. 74-82.
10. S. Wu and U. Manber, "Fast Text Searching Allowing Errors," *Comm. ACM*, Oct. 1992, pp. 83-91.
11. R. Baeza-Yates and G. Navarro, "Faster Approximate String Matching," *Algorithmica,* Vol. 23, No. 2, 1999, pp. 127-158.
12. M. Persin, "Document Filtering for Fast Ranking," *Proc. 17th Int'l ACM SIGIR Conf. Research and Development in Information Retrieval,* Springer Verlag, New York, 1994, pp. 339-348.

Nivio Ziviani *is a professor of computer science at the Federal University of Minas Gerais, Brazil. His interests include algorithms, data structures, information retrieval, Web information systems, text indexing, text searching, text compression, and related areas. He has a PhD in computer science from the University of Waterloo. Contact him at nivio@dcc.ufmg.br.*

Edleno Silva de Moura *is an assistant professor of computer science at the Federal University of Amazonas, Brazil. His interests include information retrieval, data compression, text indexing, ranking algorithms, and related areas. He has a PhD in computer science from the Federal University of Minas Gerais, Brazil. Contact him at edleno@dcc.ufmg.br.*

Gonzalo Navarro *is an assistant professor of computer science at the University of Chile. His interests include string matching, text indexing, compression, structured text, approximate searching, and related areas. He has a PhD in computer science from the University of Chile. Contact him at gnavarro@dcc.uchile.cl.*

Ricardo Baeza-Yates *is a professor of computer science at the University of Chile. His interests include information retrieval, Web information systems, digital libraries, algorithms, data structures, and related areas. He has a PhD in computer science from the University of Waterloo. Contact him at rbaeza@dcc.uchile.cl.*