Full text retrieval systems store a large corpus of natural language text and aim to provide efficient means of answering queries that involve retrieving all lexical units of a certain type (e.g. all sentences) occurring within a specified larger lexical unit (e.g. a particular chapter) and containing

- a specified word
- ... or Boolean combination of words
- ... or words within a certain distance of each other
- ... or words with a specified prefix, suffix, or stem

While the text corpus includes enough information to answer such queries without any auxiliary data structures, it is intractible to scan it all for each query, so a concordance must be used. For each word that occurs, the concordance points to all instances of that word in the text corpus. Consequently its size is commensurate with that of the main text.

The ready availability of CD ROMs has made the widespread distribution of large full text databases a practical proposition. The capacity of a single disk is approximately 550 Mbyte, the data equivalent of the 60 minute playing time of an audio CD. However, seek times for CD ROM drives are appreciably slower than those of magnetic disks, averaging 400 to 800 ms for CDs compared to 50 ms or less for typical Winchester drives. CD ROMs are sluggish because they share a standard developed for audio CDs -- which do not need fast random access. Of course, this is also why they are inexpensive, and popular. The maximum transfer rate is also set by audio requirements, at 153 kbyte/s.

The combination of high capacity and relatively low speed places great emphasis on the data structures through which databases are accessed (Cichocki & Ziemer, 1988). Moreover, it becomes crucial that these structures can be retained in main storage, for the number of disk seeks is by far the dominant factor in response time. This article explains the indexes that are required to respond quickly to the queries above. The relevant lexical units (chapters, sentences, etc.) are identified without ever reading the main text, and the concordance need be accessed just once for each word involved in the query.

Compression becomes important both for the major data structures (text, concordance) and for the smaller indexes (lexicon, disk address table) too. The

former may need to be compressed to pack a text base on to a disk, for practical usage considerations limit a database to a single disk. The latter may need to be compressed to fit into primary storage—lexicons and tables of disk pointers for a 550 Mbyte database occupy many megabytes, and keeping them on disk incurs a very heavy penalty in access time. We describe the compression potential of various data structures, but not the technical details of the compression methods themselves. (Bell *et al.*, 1990, explain techniques of text compression in general).

## Structure of a text database

In order to answer the kind of query presented above, the main text must be split into lexical units. The smallest unit (e.g. sentence) is the *grain size* of the corpus for indexing purposes. Queries about even smaller units (e.g. phrases) will have to be answered by retrieving all relevant grains (sentences) and scanning through them linearly. The choice of grain size has important implications in the time-space tradeoff. We use sentences in our examples here.

A typical lexical hierarchy comprises levels such as book, chapter, and sentence. A CD-ROM can store many hundreds of works, for a typical book occupies under 1 Mbyte, and there will often be a level—say "collection"—superior to "book."

The concordance contains, for each word, a list of pointers to its occurrences in the main text. Pointers may be expressed either as absolute word numbers or as multi-part *coordinates* in terms of the lexical hierarchy—e.g. collection, book, chapter, sentence (Choueka *et al.*, 1988). While hierarchical naming seems at first to offer more compression potential since the upper coordinates vary only slowly, in fact it is simpler to achieve good compression using absolute word numbers, and these are employed below.

Like the main text itself, the concordance will invariably be far too large to hold in primary storage. In order to access it, a word is first sought in a "lexicon" which lists all words that occur and points to their concordance entries on disk. Then the relevant block of concordance pointers can be retrieved with just one disk access.

If the query is to find all occurrences of a single word, the lexical unit identified by each pointer in the concordance entry is located, read off disk, and printed. Concordance pointers are converted to disk addresses, and this requires an index

giving the disk address of every lowest-level unit (grain). It may be necessary to display with each unit its hierarchical address, so an index to the lexical hierarchy is stored in order to convert word numbers to multi-part coordinates.

If several words are mentioned in the query, all their concordance entries are retrieved. The resulting lists of occurrence pointers are combined using the Boolean operators specified in the query. At this stage it is easy to respect constraints concerning the distance between words, since the pointers are absolute word numbers. For constraints expressed in terms of the lexical hierarchy (e.g. all occurrences of x and y in the same chapter), conversion to multi-part coordinates is necessary.

Finally, queries may involve retrieval based on incomplete word matches such as prefixes, suffices, and word stems. Words with a given prefix are adjacent in the lexicon and can easily be found. A lexicon of words spelled backwards can be used to locate those with a given suffix. However, words with a given stem cannot be found by these methods, and a technique is explained below that uses a permuted lexicon to locate partial matches. The word is expanded into all its possible completions, each is sought independently in the concordance, and the resulting lists are merged.

The principal data structures needed for these operations are:

- the main text itself;
- the concordance;
- the lexical hierarchy
- the lexicon;
- the permuted lexicon.

In order to give some indication of the size of these structures, Table 1 defines a sample text database with $10^9$ characters, or $180 \times 10^6$ words. This might contain a library of, say, 1000–1500 books. Table 2 shows, in round figures, the size of each structure. The uncompressed size is given both symbolically and numerically. The compression factors have been determined empirically for the methods mentioned below. They also are given in round figures and may vary appreciably with the kind of text. The total compressed size indicates that this database, along with concordance and other indexes, fits on a single 550 Mbyte CD-ROM with room to spare.

## The main text

This is the largest data structure. Fortunately natural language contains a great deal of redundancy and, as Table 1 shows, can be compressed to around 25% of its original bulk (for English).

### COMPRESSION POTENTIAL

In representing the main text itself, it seems at first sight that significant space could be saved—at the expense of accuracy and readability—by adopting a restricted alphabet. If all letters are mapped to lower case a 64-symbol alphabet will suffice, and if numbers are absent and punctuation is heavily restricted it may be reducible to 32 symbols, resulting in a 5-bit representation instead of the usual 8-bit one. Some of the upper-case information can be regenerated using simple algorithms, such as capitalizing letters that follow periods.

However, this apparent trade-off between quality and storage capacity is illusory. Standard compression techniques reduce the storage required for ordinary English text to between 2.0 and 2.5 bits per character—well under the 5 bits needed for the crude single-case representation (Bell *et al.*, 1990). These techniques are tailored to the adaptive compression of running text, where there is no opportunity to pre-scan the data to extract accurate statistics. Using "static" modeling in which the statistics of word occurrence are accumulated in advance gives compression at the lower end of this range. Mapping all characters to lower case increases compressibility, but by a totally insignificant amount (around 0.1%).

Although it offers no advantage in storing the main text, mapping to a single case may reduce the size of auxiliary data structures such as the lexicon. In applications which convert all retrieval terms to a single case, this will present no penalty. However, the method that we use for compressing the main text is based on the occurrence frequency of individual words, and for this a complete lexicon is necessary. Moreover, a significant proportion of the capitalized words do not appear in uncapitalized form (e.g. proper names), which means that the reduction in lexicon size is less than might be expected.

However, it *is* important to reduce lexicon size by not including punctuation (periods, commas, quotation marks, etc.) as part of words but instead treating punctuation sequences as words in their own right. For example, this was found to reduce the vocabulary of the Brown Corpus, a standard million-word corpus of American English (Kucera & Francis, 1967), from 100,000 to 58,000 words (Witten & Bell, 1990).

### GRAIN SIZE

Grain size affects the compression of the main text. Good compression methods are stream-oriented, and treating units individually incurs a small overhead in terminating compression at the end of each one and re-starting it for the next. With the compression method that we use (arithmetic coding; see Witten *et al.*, 1987), termination incurs a 2-bit overhead.

Further overhead stems from the resolution of disk addressing. In general, coarser resolution gives smaller pointers but more wasted space on disk. Byte addressing incurs an additional byte-padding overhead of 4 bits on average. If bit addressing is used, no padding is needed—but disk pointers are 3 bits larger. Despite its small disadvantage, byte addressing is preferred on grounds of simplicity.

Combining these two sources of overhead adds an average of 6 bits to each grain. A typical 20-word sentence occupies 220 bits in compressed form, so for sentence-sized grains the overhead is only 3%. Larger granularity reduces overhead at the expense of processing time, since units must be completely decoded to provide access to their constituents.

A more significant overhead is incurred by the index that associates grains with disk addresses, discussed below.

## The concordance

The concordance is by far the largest auxiliary data structure. For each word in the text, a pointer to it appears in the concordance. Since pointers are word numbers, and 28 bits are needed to address 180 million words, approximately 3 bytes are required for each. Since words themselves average 5 or 6 characters in length

(including a terminating space), the concordance, uncompressed, occupies about half the size of the uncompressed text.

Because it is so large, the concordance is often reduced by omitting from it a list of common words (often called the "stop list"). Common words naturally have the largest concordance entries, and, by the same token, are least likely to be found useful in queries. This can sometimes damage performance—Klein *et al.* (1989) mention a system that received a request to locate the phrase "un de ces" in the works of André Gide but could not answer it because all three words were on the stop list!

If the concordance is properly compressed it is these common words whose entries will be reduced most. For example, we have found that the most common word in a particular corpus, "the," accounts for 7.6% of all concordance entries but only 0.34% of compressed concordance size. The 100 most common words account for 76% of references but 44% of compressed size. It is not clear that the gain of omitting words on a stop list is worth the cost in terms of reduced functionality.

We now turn to compression of the concordance. Consider the occurrences of a particular word. Their number is known: it is just the occurrence count, and this is stored in the word's lexicon entry (as described below). Dividing this by the size of the text gives the word's occurrence probability $p$. A simple theoretical model can be used to predict the distribution of inter-word gaps, and this distribution can be used to encode the sequence of concordance pointers efficiently. We have found that this compresses concordance pointers to under 50% of their original size.

As Table 2 shows, in round terms the uncompressed concordance is half the size of the original text, but the compressed text and concordance together occupy half the space of the original text.

## The lexical hierarchy

The purpose of the lexical hierarchy is two-fold. First, it enables absolute word numbers obtained from the concordance to be converted into multi-part coordinates so that a query can be restricted to certain lexical units like a book or chapter. Second, it provides the disk address of each individual grain so that its text—and the text of any higher-level unit—can be located. Since these operations must be

performed quickly, the lexical hierarchy information should reside in main store if possible.

### STORING THE LEXICAL HIERARCHY

Figure 1 illustrates a suitable structure to record the lexical hierarchy of the text. The lowest level records the disk address and word number of each sentence. This is a large data structure, because 30 bits are needed to address individual bytes of a CD-ROM ($2^{30} \approx 1$ billion), nearly as many bits are needed to represent word numbers, and there may be 9 million sentences—perhaps 72 Mbyte in total. Compression will certainly be necessary to fit this into the main store of a personal computer.

Instead of absolute disk addresses and word numbers for each sentence, it is more economical to store

- the number of bytes a sentence occupies on disk;
- the number of words in it.

These two quantities are obviously highly correlated, and advantage can be taken of this in compression. In effect this means that, since disk pointers are needed anyway, little further space is required for the word numbers.

### COMPRESSING THE LOWEST-LEVEL INDEX

Compression may only be worthwhile for the lowest level of the lexical hierarchy because the exponential growth in index size makes gains at higher levels relatively insignificant. Figure 1 shows the "sentence" index divided into fixed-length blocks. Each is headed by the word number and disk address of the first unit in the block, stored literally, and subsequent entries are represented as increments from the preceding one. Consider the number of words in the sentence. Since there are probably only a few dozen different sentence lengths, and the occurrence of them is highly clustered (few under 4; few over 40), they can be compressed very effectively. Our experiments indicate that less than 6 bits on average are needed to represent sentence lengths.

The number of words in a sentence can be used to predict the number of bytes it occupies. The discrepancy between the prediction and the actual size can then be coded. Our experiments with sentence-sized grains indicate that under 4 bits are

needed on average to represent the prediction error. Thus the number of bytes occupied can be derived from a quantity that occupies just a few bits, instead of a full 30-bit disk address.

Less than 10 bits in total are needed for each unit at the lowest lexical level. The word number and disk address of the first unit in each block, shown in Figure 1, will add some overhead to this. If there are an average of 200 sentences per chapter, each compressed block will occupy 250 bytes, and the overhead will amount to around 3%. Furthermore, storing the chapter (and book) index may double this figure. As Table 2 shows, the total size of the compressed lexical hierarchy amounts to perhaps 12 Mbyte—a substantial improvement on the uncompressed size of 72 Mbyte.

Nevertheless, 12 Mbyte will tax the main store of many personal computers, and it may be necessary to keep the lowest level of the lexical hierarchy on disk. If the structure illustrated in Figure 1 is reorganized slightly by moving the word number of the first sentence of each chapter from the sentence index to the chapter index, then only one additional disk access will be needed every time a word number is converted into a hierarchy coordinate, or every time a disk address is sought. Then the largest main memory data structure will be the index for the second-lowest level of the lexical hierarchy ("chapter" in Figure 1), and it might conceivably be worth compressing this. Such tactics can be used to tune a particular database to a particular target computer configuration.

## The lexicon

In order to access the concordance, a lexicon is used which contains an entry for every different word that occurs in the main text. The entry records:

- the word itself;
- a pointer to its concordance entry.

When responding to user requests to locate all occurrences of a particular word, the lexicon is accessed via the word to locate its entry in the concordance, which is then decoded and the appropriate units of text read off.

The lexicon may also be used for another purpose: decoding the main text. If the compression method uses the statistics of word occurrence for encoding purposes, then each word must have

- its occurrence count

stored too. When decoding the main text, the lexicon will be accessed via the occurrence count. Since the lexicon is used for every word decoded, it is essential that access be fast. Consequently it is worthwhile storing it in uncompressed form if possible.

The average word length in an English lexicon is likely to be around 8 characters, including terminator. (Note that this is different from the average word length in text, where short words are used more often.) The concordance is a large data structure and pointers into it need to be something approaching a full disk address. Occurrence counts may need 3 bytes. Consequently each lexicon entry is likely to occupy an average of 15 bytes. Uncompressed, a 100,000-word lexicon will occupy around 1.5 Mbyte.

Advantage can be taken of the structure of the lexicon to reduce this considerably. Since words are stored in alphabetical order, each is likely to share a prefix with its predecessor, and only the length of this prefix need be stored (Figure 2). Coupled with simple compression of suffixes leads to a storage requirement of just over 2 bytes per word. Most occurrence counts are small, and this fact can be used to reduce the space required for a count to around 5 bits. Finally, the size of a concordance entry for a word can, not surprisingly, be predicted from the word's occurrence count—not exactly, since the concordance is compressed, but to a good approximation. The error in the prediction can be coded in around 6 bits. This leads to a requirement of just over 3 bytes per lexicon entry, so that now the 100,000-word lexicon occupies only 300 Kbyte of primary storage—but at the expense of greatly increased decoding time.

## The permuted lexicon

There is an elegant method for finding query terms of the forms $X$, $X*$, $*X$ and $*X*$, where $X$ is a specified string of characters and * matches any string (Bratley and Choueka, 1982). It employs a permuted dictionary in which each word appears

in all possible rotated positions. For example, the word "hello" will appear four times, as "o/hell", "lo/hel", "llo/he", and "ello/h". Because the permuted dictionary is used as an adjunct to, and not a replacement for, the lexicon, the form "/hello" is not stored in it.

It is a remarkable fact that all query terms of the forms above can be expanded very efficiently using this method. As an example, Figure 3 shows a permuted version of the lexicon of Figure 2. Just as all words of the form abas* can be found easily by a binary search of the sorted list at the left of Figure 2, all words of the form *on* (i.e. "abalone" and "abandon") can be found by a binary search seeking entries beginning with the pattern "on" in Figure 3 (middle of third column), and all words containing "s" (i.e. abacus, abase and abash) appear together under "s". Nothing further need be stored since once the words are obtained their concordance entries can be retrieved from the lexicon.

The number of words in the permuted dictionary can be calculated as the number of words in the lexicon times the average length of each less 1. A 100,000 word lexicon with average length 8 characters (including terminator) will generate 700,000 permuted words, which occupy perhaps 8 Mbyte in uncompressed form (for the average length of a word in the permuted dictionary is greater than the average length in the original lexicon, since long words have more permutations). Compression, using the same technique as for words in the lexicon, reduces this to around 2.5 Mbyte.

## Access time

With the indexing structure that has been presented, the procedure of finding and displaying all sentences in the sample text that contain a given word involves

- seeking the word in the lexicon;
- accessing the disk to read the block of concordance entries;
- decoding the block of concordance entries;
- converting each concordance pointer to a disk address using the lexical hierarchy index;
- reading, decoding and displaying the sentences at the disk addresses.

Assuming that the lexical hierarchy can be kept in main store, this involves just one extra disk access (to read the concordance entries) over and above the inevitable disk access for every sentence printed.

It also involves a fair amount of decoding of compressed information. If the lexicon is compressed, it will need to be blocked so that the block containing the target word can be located by binary searching, then decoded and searched linearly for the word itself. The concordance entries will have to be decoded. In order to convert each word number to a disk address, the lexical hierarchy index will need to be consulted and, once the relevant block is located (grey region of Figure 1), it will be decoded and searched linearly. Finally, the sentences which are retrieved must be decoded and, if word compression is used, this involves consulting the lexicon for every word printed.

## Conclusion

The designer of a full-text database for CD-ROM must pay careful attention to the indexing structures through which the text is accessed. The use of suitable compression techniques substantially increases the volume of text that can be held on disk, and substantially decreases the amount of primary storage needed to hold the indexes when processing queries. Apparent economies such as folding text to a single case or omitting commonly-used words from the concordance may not be worthwhile when proper compression methods are used. When it comes to detailed design of the data structures involved (e.g. block sizes), there will be an intricate tradeoff between disk space consumed, main memory required, disk seek time, processor speed, and desired response time.

## Acknowledgements

We gratefully acknowledge enlightening discussions with Dr Choueka and Dr Klein. This research is supported by the Natural Sciences and Engineering Research Council of Canada.

## References

Bell, T.C., Cleary, J.G. and Witten, I.H. (1990) *Text compression.* Prentice Hall, Englewood Cliffs, NJ.

Bratley, P. and Choueka, Y. (1982) "Processing truncated terms in document retrieval systems," *Information Processing and Management 18*(5): 257–266.

Choueka, Y., Fraenkel, A.S. and Klein, S.T. (1988) "Compression of concordances in full-text retrieval systems," *Proc 11th Conference on Research and Development in Information Systems*: 597–612, Grenoble; June.

Cichocki, E.M. and Ziemer, S.M. (1988) "Design considerations for CD-ROM retrieval software," *J. American Society for Information Science 39*(1): 43–46; January.

Klein, S.T., Bookstein, A. and Deerwester, S. (1989) "Storing text retrieval systems on CD-ROM: compression and encryption considerations," *ACM Trans. Information Systems 7*(3): 230–245; July.

Kucera, H. and Francis, W.N. (1967) "Computational analysis of present-day American English," Brown University Press, Providence, RI.

Witten, I.H. and Bell, T.C. (1990) "Source models for natural language text," *Int. J. Man-Machine Studies 32*(5): 545–579; May.

Witten, I.H., Neal, R. and Cleary, J.G. (1987) "Arithmetic coding for data compression," *Communications of the ACM 30*(6): 520–540; June.

## Table and Figure captions

Table 1 Size of the example text base

Table 2 Principal data structures and their sizes

Figure 1 Storing the lexical hierarchy and disk addresses

Figure 2 Part of a lexicon and its front compression coding

Figure 3 Permuted version of Figure 2

| | | |
|---|---|---|
| Characters | $c$ | $1000 \times 10^6$ |
| Words | $w$ | $180 \times 10^6$ |
| Sentences | $s$ | $9 \times 10^6$ |
| Vocabulary | $v$ | 100,000 |

Table 1 Size of the example text base

| | Uncompressed size | | Compressed size | |
|---|---|---|---|---|
| | (byte) | (Mbyte) | factor | (Mbyte) |
| Main text | $c$ | 1000 | 25% | 250 |
| Concordance | $c/2$ | 500 | 50% | 250 |
| Lexical hierarchy | $8s$ | 72 | 15% | 12 |
| Lexicon | $15v$ | 1.5 | 20% | 0.3 |
| Permuted lexicon | $80v$ | 8 | 33% | 2.5 |
| | | | *total* | 515 Mbyte |

Table 2 Principal data structures and their sizes

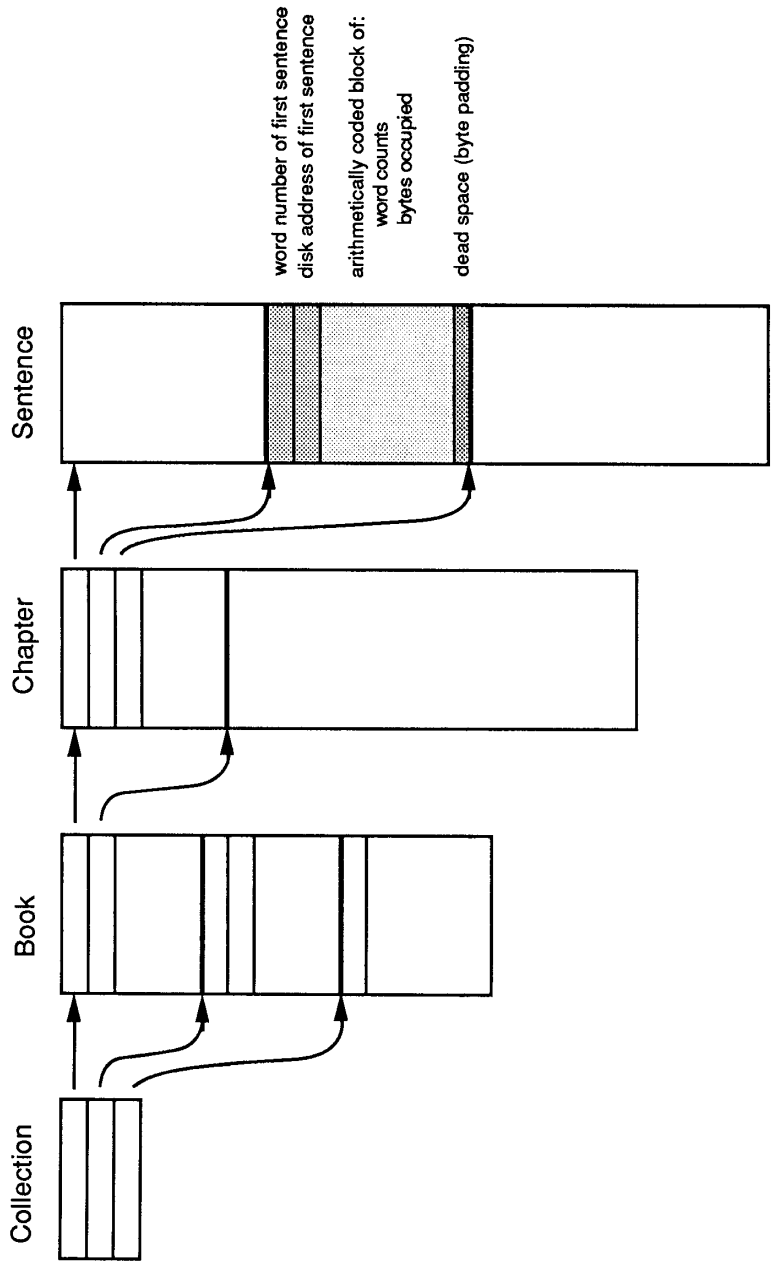
word number of first sentence
disk address of first sentence
arithmetically coded block of:
word counts
bytes occupied
dead space (byte padding)
Sentence
Chapter
Book
Collection

Figure 1

| | |
|---|---|
| a | a |
| aback | 1back |
| abacus | 4us |
| abalone | 3lone |
| abandon | 3ndon |
| abase | 3se |
| abash | 4h |
| abate | 3te |

Figure 2

| | | |
|---|---|---|
| ack/ab | base/a | lone/aba |
| acus/ab | bash/a | n/abando |
| alone/ab | bate/a | ndon/aba |
| andon/ab | ck/aba | ne/abalo |
| ase/ab | cus/aba | on/aband |
| ash/ab | don/aban | one/abal |
| ate/ab | e/abalon | s/abacu |
| back/a | e/abas | se/aba |
| bacus/a | e/abat | sh/aba |
| balone/a | h/abas | te/aba |
| bandon/a | k/abac | us/abac |

Figure 3